Chapter 2

ELEMENTARY DATA STRUCTURES

Now that we have examined the fundamental methods we need to express and analyze algorithms, we might feel all set to begin. But, alas, we need to make one last diversion, and that is a discussion of data structures. One of the basic techniques for improving algorithms is to structure the data in such a way that the resulting operations can be efficiently carried out. In this chapter, we review only the most basic and commonly used data structures. Many of these are used in subsequent chapters. We should be familiar with stacks and queues (Section 2.1), binary trees (Section 2.2), and graphs (Section 2.6) and be able to refer to the other structures as needed.

2.1 STACKS AND QUEUES

One of the most common forms of data organization in computer programs is the ordered or linear list, which is often written as $a = (a_1, a_2, \ldots, a_n)$. The a_i's are referred to as *atoms* and they are chosen from some set. The null or empty list has $n = 0$ elements. A *stack* is an ordered list in which all insertions and deletions are made at one end, called the *top*. A *queue* is an ordered list in which all insertions take place at one end, the *rear*, whereas all deletions take place at the other end, the *front*.

The operations of a stack imply that if the elements A, B, C, D, and E are inserted into a stack, in that order, then the first element to be removed (deleted) must be E. Equivalently we say that the last element to be inserted into the stack is the first to be removed. For this reason stacks are sometimes referred to as **L**ast **I**n **F**irst **O**ut (**LIFO**) lists. The operations of a queue require that the first element that is inserted into the queue is the first one to be removed. Thus queues are known as **F**irst **I**n **F**irst **O**ut (**FIFO**) lists. See Figure 2.1 for examples of a stack and a queue each containing the same

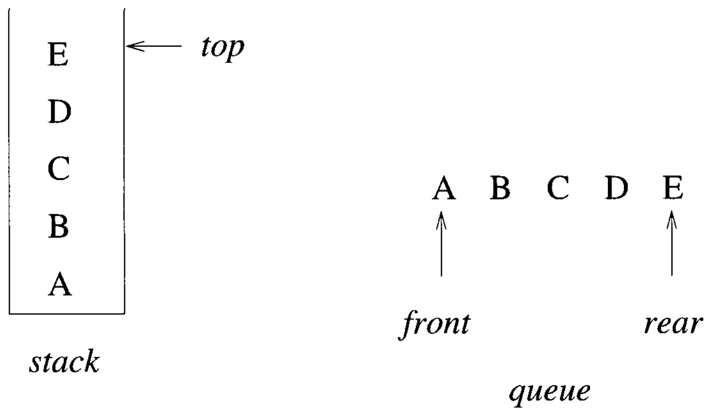


Figure 2.1 Example of a stack and a queue

five elements inserted in the same order. Note that the data object queue as defined here need not correspond to the concept of queue that is studied in queuing theory.

The simplest way to represent a stack is by using a one-dimensional array, say $stack[0 : n - 1]$, where n is the maximum number of allowable entries. The first or bottom element in the stack is stored at $stack[0]$, the second at $stack[1]$, and the ith at $stack[i-1]$. Associated with the array is a variable, typically called *top*, which points to the top element in the stack. To test whether the stack is empty, we ask "**if** $(top < 0)$". If not, the topmost element is at $stack[top]$. Checking whether the stack is full can be done by asking "**if** $(top \geq n - 1)$". Two more substantial operations are inserting and deleting elements. The corresponding algorithms are Add and Delete (Algorithm 2.1).

Each execution of Add or Delete takes a constant amount of time and is independent of the number of elements in the stack.

Another way to represent a stack is by using links (or pointers). A *node* is a collection of data and link information. A stack can be represented by using nodes with two fields, possibly called *data* and *link*. The data field of each node contains an item in the stack and the corresponding link field points to the node containing the next item in the stack. The link field of the last node is zero, for we assume that all nodes have an address greater than zero. For example, a stack with the items A, B, C, D, and E inserted in that order, looks as in Figure 2.2.

```
Algorithm Add(item)
// Push an element onto the stack. Return true if successful;
// else return false. item is used as an input.
{
    if ($top \geq n - 1$) then
    {
        write ("Stack is full!"); return false;
    }
    else
    {
        $top := top + 1$; $stack[top] := item$; return true;
    }
}
```

```
Algorithm Delete(item)
// Pop the top element from the stack. Return true if successful
// else return false. item is used as an output.
{
    if ($top < 0$) then
    {
        write ("Stack is empty!"); return false;
    }
    else
    {
        $item := stack[top]$; $top := top - 1$; return true;
    }
}
```

Algorithm 2.1 Operations on a stack

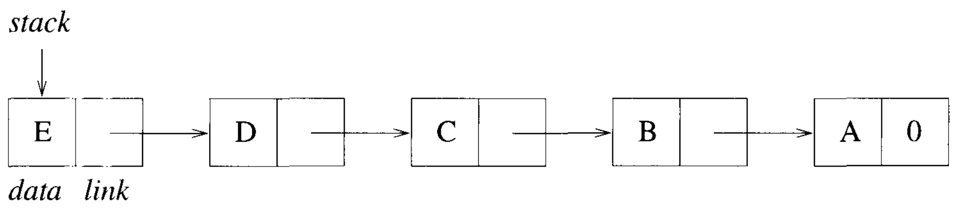


Figure 2.2 Example of a five-element, linked stack

```
// Type is the type of data.
node =record
{
    Type data; node *link;
}

Algorithm Add(item)
{
    // Get a new node.
    temp := new node;
    if (temp ≠ 0) then
    {
        (temp → data) := item; (temp → link) := top;
        top := temp; return true;
    }
    else
    {
        write ("Out of space!");
        return false;
    }
}

Algorithm Delete(item)
{
    if (top = 0) then
    {
        write ("Stack is empty!");
        return false;
    }
    else
    {
        item := (top → data); temp := top;
        top := (top → link);
        delete temp; return true;
    }
}
```

Algorithm 2.2 Link representation of a stack

The variable *top* points to the topmost node (the last item inserted) in the list. The empty stack is represented by setting $top := 0$. Because of the way the links are pointing, insertion and deletion are easy to accomplish. See Algorithm 2.2.

In the case of Add, the statement *temp* := **new** *node*; assigns to the variable *temp* the address of an available node. If no more nodes exist, it returns 0. If a node exists, we store appropriate values into the two fields of the node. Then the variable *top* is updated to point to the new top element of the list. Finally, **true** is returned. If no more space exists, it prints an error message and returns **false**. Refering to Delete, if the stack is empty, then trying to delete an item produces the error message "Stack is empty!" and **false** is returned. Otherwise the top element is stored as the value of the variable *item*, a pointer to the first node is saved, and *top* is updated to point to the next node. The deleted node is returned for future use and finally **true** is returned.

The use of links to represent a stack requires more storage than the sequential array $stack[0 : n - 1]$ does. However, there is greater flexibility when using links, for many structures can simultaneously use the same pool of available space. Most importantly the times for insertion and deletion using either representation are independent of the size of the stack.

An efficient queue representation can be obtained by taking an array $q[0 : n - 1]$ and treating it as if it were circular. Elements are inserted by increasing the variable *rear* to the next free position. When $rear = n - 1$, the next element is entered at $q[0]$ in case that spot is free. The variable *front* always points one position counterclockwise from the first element in the queue. The variable $front = rear$ if and only if the queue is empty and we initially set $front := rear := 0$. Figure 2.3 illustrates two of the possible configurations for a circular queue containing the four elements J1 to J4 with $n > 4$.

To insert an element, it is necessary to move *rear* one position clockwise. This can be done using the code

```
if ($rear = n - 1$) then $rear := 0$;
else $rear := rear + 1$;
```

A more elegant way to do this is to use the built-in modulo operator which computes remainders. Before doing an insert, we increase the rear pointer by saying $rear := (rear + 1)$ **mod** n;. Similarly, it is necessary to move *front* one position clockwise each time a deletion is made. An examination of Algorithm 2.3(a) and (b) shows that by treating the array circularly, addition and deletion for queues can be carried out in a fixed amount of time or $O(1)$.

One surprising feature in these two algorithms is that the test for queue full in AddQ and the test for queue empty in DeleteQ are the same. In the

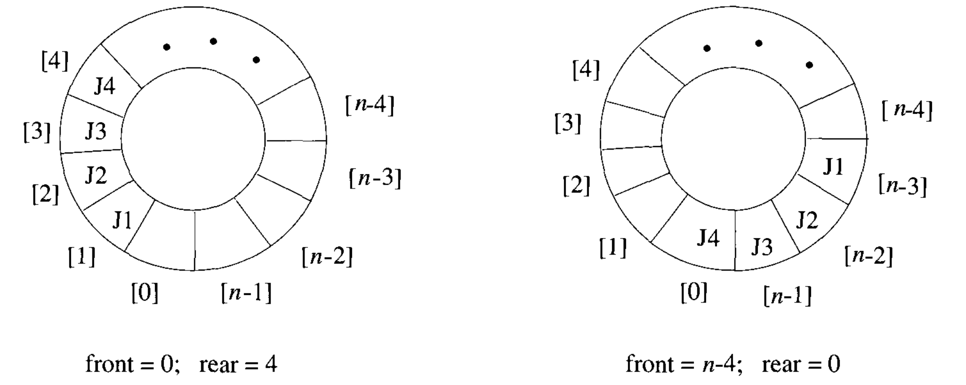


Figure 2.3 Circular queue of capacity $n - 1$ containing four elements J1, J2, J3, and J4

case of AddQ, however, when $front = rear$, there is actually one space free, $q[rear]$, since the first element in the queue is not at $q[front]$ but is one position clockwise from this point. However, if we insert an item there, then we cannot distinguish between the cases full and empty, since this insertion leaves $front = rear$. To avoid this, we signal queue full and permit a maximum of $n - 1$ rather than n elements to be in the queue at any time. One way to use all n positions is to use another variable, *tag*, to distinguish between the two situations; that is, $tag = 0$ if and only if the queue is empty. This however slows down the two algorithms. Since the AddQ and DeleteQ algorithms are used many times in any problem involving queues, the loss of one queue position is more than made up by the reduction in computing time.

Another way to represent a queue is by using links. Figure 2.4 shows a queue with the four elements A, B, C, and D entered in that order. As with the linked stack example, each node of the queue is composed of the two fields *data* and *link*. A queue is pointed at by two variables, *front* and *rear*. Deletions are made from the front, and insertions at the rear. Variable $front = 0$ signals an empty queue. The procedures for insertion and deletion in linked queues are left as exercises.

EXERCISES

1. Write algorithms for AddQ and DeleteQ, assuming the queue is represented as a linked list.

```
Algorithm AddQ(item)
// Insert item in the circular queue stored in $q[0 : n - 1]$.
// rear points to the last item, and front is one
// position counterclockwise from the first item in q.
{
    $rear := (rear + 1)$ mod $n$; // Advance rear clockwise.
    if ($front = rear$) then
    {
        write ("Queue is full!");
        if ($front = 0$) then $rear := n - 1$;
        else $rear := rear - 1$;
        // Move rear one position counterclockwise.
        return false;
    }
    else
    {
        $q[rear] := item$; // Insert new item.
        return true;
    }
}
```

(a) Addition of an element

```
Algorithm DeleteQ(item)
// Removes and returns the front element of the queue $q[0 : n - 1]$.
{
    if ($front = rear$) then
    {
        write ("Queue is empty!");
        return false;
    }
    else
    {
        $front := (front + 1)$ mod $n$; // Advance front clockwise.
        $item := q[front]$; // Set item to front of queue.
        return true;
    }
}
```

(b) Deletion of an element

Algorithm 2.3 Basic queue operations

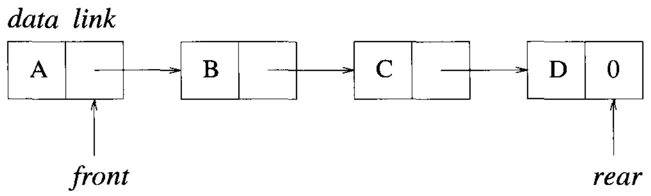


Figure 2.4 A linked queue with four elements

2. A linear list is being maintained circularly in an array $c[0 : n-1]$ with f and r set up as for circular queues.
   (a) Obtain a formula in terms of f, r, and n for the number of elements in the list.
   (b) Write an algorithm to delete the kth element in the list.
   (c) Write an algorithm to insert an element y immediately after the kth element.

   What is the time complexity of your algorithms for parts (b) and (c)?

3. Let $X = (x_1, \ldots, x_n)$ and $Y = (y_1, \ldots, y_m)$ be two linked lists. Write an algorithm to merge the two lists to obtain the linked list $Z = (x_1, y_1, x_2, y_2, \ldots, x_m, y_m, x_{m+1}, \ldots, x_n)$ if $m \leq n$ or $Z = (x_1, y_1, x_2, y_2, \ldots, x_n, y_n, y_{n+1}, \ldots, y_m)$ if $m > n$.

4. A double-ended queue (deque) is a linear list for which insertions and deletions can occur at either end. Show how to represent a deque in a one-dimensional array and write algorithms that insert and delete at either end.

5. Consider the hypothetical data object $X2$. The object $X2$ is a linear list with the restriction that although additions to the list can be made at either end, deletions can be made from one end only. Design a linked list representation for $X2$. Specify initial and boundary conditions for your representation.

2.2 TREES

Definition 2.1 [Tree] A *tree* is a finite set of one or more nodes such that there is a specially designated node called the *root* and the remaining nodes are partitioned into $n \geq 0$ disjoint sets $T_1, \ldots, T_n$, where each of these sets is a tree. The sets $T_1, \ldots, T_n$ are called the *subtrees* of the root. □

2.2.1 Terminology

There are many terms that are often used when referring to trees. Consider the tree in Figure 2.5. This tree has 13 nodes, each data item of a node being a single letter for convenience. The root contains A (we usually say node A), and we normally draw trees with their roots at the top. The number of subtrees of a node is called its *degree*. The degree of A is 3, of C is 1, and of F is 0. Nodes that have degree zero are called *leaf* or *terminal* nodes. The set {K, L, F, G, M, I, J} is the set of leaf nodes of Figure 2.5. The other nodes are referred to as *nonterminals*. The roots of the subtrees of a node X are the *children* of X. The node X is the *parent* of its children. Thus the children of D are H, I, and J, and the parent of D is A.

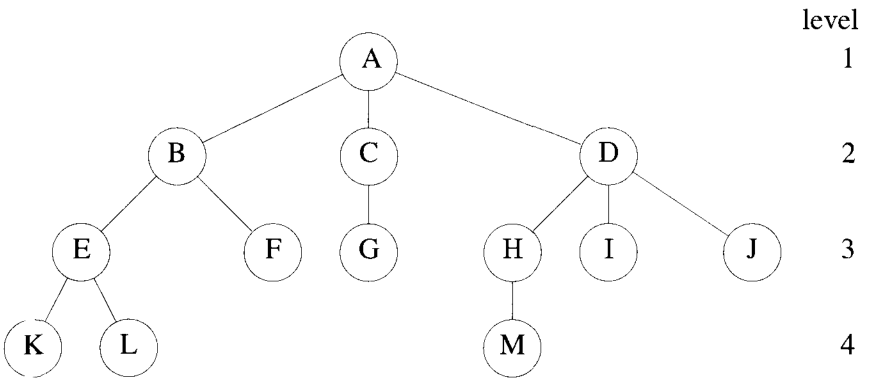


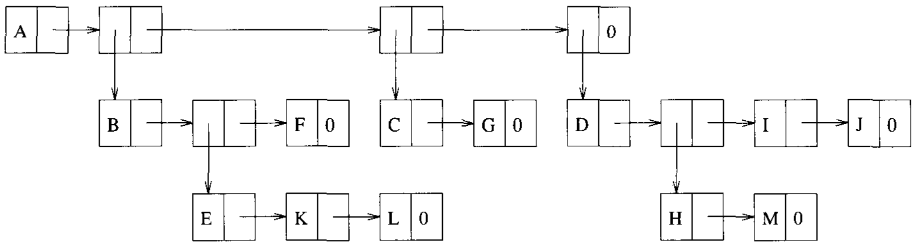
Figure 2.5 A sample tree

Children of the same parent are said to be *siblings*. For example H, I, and J are siblings. We can extend this terminology if we need to so that we can ask for the grandparent of M, which is D, and so on. The *degree* of a tree is the maximum degree of the nodes in the tree. The tree in Figure 2.5 has degree 3. The *ancestors* of a node are all the nodes along the path from the root to that node. The ancestors of M are A, D, and H.

The *level* of a node is defined by initially letting the root be at level one. If a node is at level p, then its children are at level $p + 1$. Figure 2.5 shows the levels of all nodes in that tree. The *height* or *depth* of a tree is defined to be the maximum level of any node in the tree.

A *forest* is a set of $n \geq 0$ disjoint trees. The notion of a forest is very close to that of a tree because if we remove the root of a tree, we get a forest. For example, in Figure 2.5 if we remove A, we get a forest with three trees.

Now how do we represent a tree in a computer's memory? If we wish to use a linked list in which one node corresponds to one node in the tree, then a node must have a varying number of fields depending on the number of children. However, it is often simpler to write algorithms for a data representation in which the node size is fixed. We can represent a tree using a fixed node size list structure. Such a list representation for the tree of Figure 2.5 is given in Figure 2.6. In this figure nodes have three fields: *tag*, *data*, and *link*. The fields *data* and *link* are used as before with the exception that when $tag = 1$, *data* contains a pointer to a list rather than a data item. A tree is represented by storing the root in the first node followed by nodes that point to sublists each of which contains one subtree of the root.



The *tag* field of a node is one if it has a down-pointing arrow; otherwise it is zero.

Figure 2.6 List representation for the tree of Figure 2.5

2.2.2 Binary Trees

A binary tree is an important type of tree structure that occurs very often. It is characterized by the fact that any node can have at most two children; that is, there is no node with degree greater than two. For binary trees we distinguish between the subtree on the left and that on the right, whereas for other trees the order of the subtrees is irrelevant. Furthermore a binary tree is allowed to have zero nodes whereas any other tree must have at least one node. Thus a binary tree is really a different kind of object than any other tree.

Definition 2.2 A *binary tree* is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called the *left* and *right* subtrees. □

Figure 2.7 shows two sample binary trees. These two trees are special kinds of binary trees. Figure 2.7(a) is a *skewed* tree, skewed to the left.

There is a corresponding tree skewed to the right, which is not shown. The tree in Figure 2.7(b) is called a *complete* binary tree. This kind of tree is defined formally later on. Notice that for this tree all terminal nodes are on two adjacent levels. The terms that we introduced for trees, such as degree, level, height, leaf, parent, and child, all apply to binary trees in the same way.

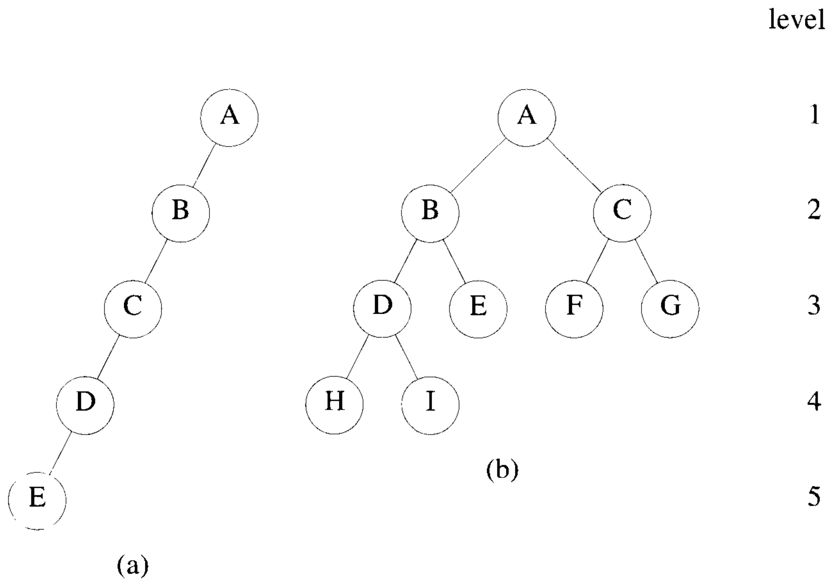


Figure 2.7 Two sample binary trees

Lemma 2.1 The maximum number of nodes on level i of a binary tree is 2^{i-1}. Also, the maximum number of nodes in a binary tree of depth k is $2^k - 1, k > 0$. □

The binary tree of depth k that has exactly $2^k - 1$ nodes is called a *full* binary tree of depth k. Figure 2.8 shows a full binary tree of depth 4. A very elegant sequential representation for full binary trees results from sequentially numbering the nodes, starting with the node on level one, then going to those on level two, and so on. Nodes on any level are numbered from left to right (see Figure 2.8). A binary tree with n nodes and depth k is *complete* iff its nodes correspond to the nodes that are numbered one to n in the full binary tree of depth k. A consequence of this definition is that in a complete tree, leaf nodes occur on at most two adjacent levels. The nodes of an n-node complete tree may be compactly stored in a one-dimensional array, $tree[1 : n]$, with the node numbered i being stored in $tree[i]$. The next lemma shows how to easily determine the locations of the parent, left child, and right child of any node i in the binary tree without explicitly storing any link information.

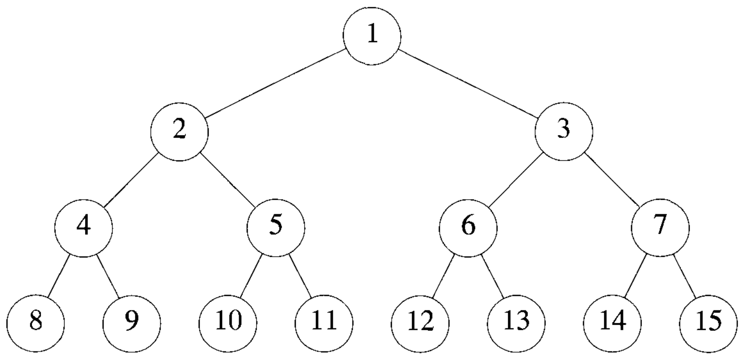


Figure 2.8 Full binary tree of depth 4

Lemma 2.2 If a complete binary tree with n nodes is represented sequentially as described before, then for any node with index i, $1 \leq i \leq n$, we have:

1. $parent(i)$ is at $\lfloor i/2 \rfloor$ if $i \neq 1$. When $i = 1, i$ is the root and has no parent.
2. $lchild(i)$ is at $2i$ if $2i \leq n$. If $2i > n$, i has no left child.
3. $rchild(i)$ is at $2i + 1$ if $2i + 1 \leq n$. If $2i + 1 > n$, i has no right child. □

This representation can clearly be used for all binary trees though in most cases there is a lot of unutilized space. For complete binary trees the representation is ideal as no space is wasted. For the skewed tree of Figure 2.7, however, less than a third of the array is utilized. In the worst case a right-skewed tree of depth k requires $2^k - 1$ locations. Of these only k are occupied.

Although the sequential representation, as in Figure 2.9, appears to be good for complete binary trees, it is wasteful for many other binary trees. In addition, the representation suffers from the general inadequacies of sequential representations. Insertion or deletion of nodes requires the movement

tree	A	B	-	C	-	-	-	D	-	. . .	E
tree	A	B	C	D	E	F	G	H	I		
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	. . .	(16)

Figure 2.9 Sequential representation of the binary trees of Figure 2.7

of potentially many nodes to reflect the change in level number of the remaining nodes. These problems can be easily overcome through the use of a linked representation. Each node has three fields: *lchild*, *data*, and *rchild*. Although this node structure makes it difficult to determine the parent of a node, for most applications it is adequate. In case it is often necessary to be able to determine the parent of a node, then a fourth field, *parent*, can be included with the obvious interpretation. The representation of the binary trees of Figure 2.7 using a three-field structure is given in Figure 2.10.

2.3 DICTIONARIES

An abstract data type that supports the operations insert, delete, and search is called a *dictionary*. Dictionaries have found application in the design of numerous algorithms.

Example 2.1 Consider the database of books maintained in a library system. When a user wants to check whether a particular book is available, a *search* operation is called for. If the book is available and is issued to the user, a *delete* operation can be performed to remove this book from the set of available books. When the user returns the book, it can be *insert*ed back into the set. □

It is essential that we are able to support the above-mentioned operations as efficiently as possible since these operations are performed quite frequently. A number of data structures have been devised to realize a dictionary. At a very high level these can be categorized as comparison methods and direct access methods. Hashing is an example of the latter. We elaborate only on binary search trees which are an example of the former.

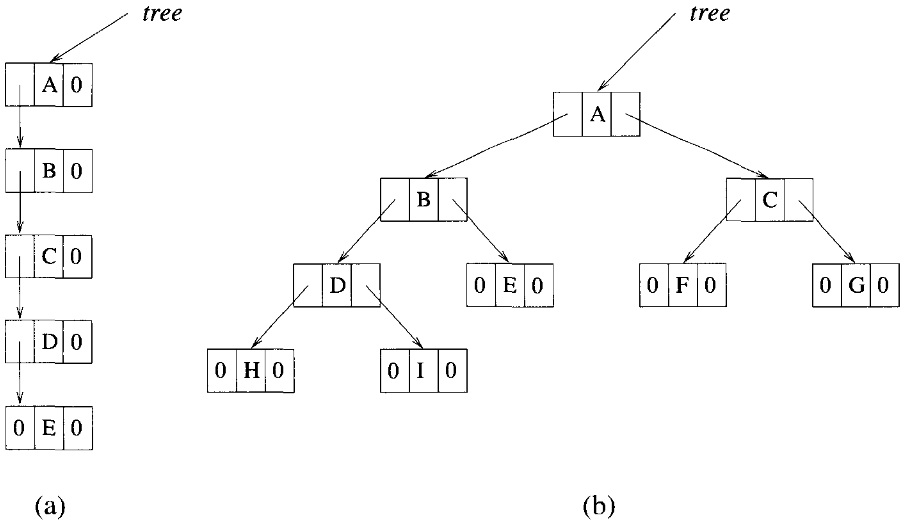


Figure 2.10 Linked representations for the binary trees of Figure 2.7

2.3.1 Binary Search Trees

Definition 2.3 [Binary search tree] A *binary search tree* is a binary tree. It may be empty. If it is not empty, then it satisfies the following properties:

1. Every element has a key and no two elements have the same key (i.e., the keys are distinct).
2. The keys (if any) in the left subtree are smaller than the key in the root.
3. The keys (if any) in the right subtree are larger than the key in the root.
4. The left and right subtrees are also binary search trees. □

A *binary search tree* can support the operations search, insert, and delete among others. In fact, with a binary search tree, we can search for a data element both by key value and by rank (i.e., find an element with key x, find the fifth-smallest element, delete the element with key x, delete the fifth-smallest element, insert an element and determine its rank, and so on).

There is some redundancy in the definition of a binary search tree. Properties 2, 3, and 4 together imply that the keys must be distinct. So, property 1 can be replaced by the property: The root has a key.

Some examples of binary trees in which the elements have distinct keys are shown in Figure 2.11. The tree of Figure 2.11(a) is not a binary search tree, despite the fact that it satisfies properties 1, 2, and 3. The right subtree fails to satisfy property 4. This subtree is not a binary search tree, as its right subtree has a key value (22) that is smaller than that in the subtree's root (25). The binary trees of Figure 2.11(b) and (c) are binary search trees.

Searching a Binary Search Tree

Since the definition of a binary search tree is recursive, it is easiest to describe a recursive search method. Suppose we wish to search for an element with key x. An element could in general be an arbitrary structure that has as one of its fields a *key*. We assume for simplicity that the element just consists of a *key* and use the terms element and key interchangeably. We begin at the root. If the root is 0, then the search tree contains no elements and the search is unsuccessful. Otherwise, we compare x with the key in the root. If x equals this key, then the search terminates successfully. If x is less than the key in the root, then no element in the right subtree can have key value x, and only the left subtree is to be searched. If x is larger than the key in the root, only the right subtree needs to be searched. The subtrees can be searched recursively as in Algorithm 2.4. This function assumes a linked

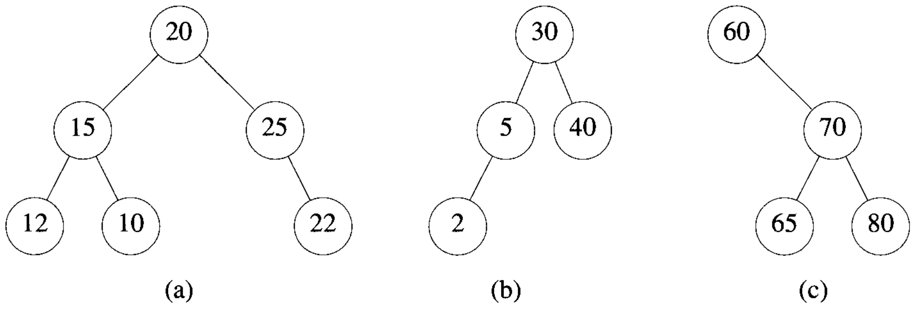


Figure 2.11 Binary trees

representation for the search tree. Each node has the three fields *lchild*, *rchild*, and *data*. The recursion of Algorithm 2.4 is easily replaced by a **while** loop, as in Algorithm 2.5.

```
Algorithm Search($t, x$)
{
    if ($t = 0$) then return 0;
    else if ($x = t \to data$) then return $t$;
        else if ($x < t \to data$) then
                return Search($t \to lchild, x$);
            else return Search($t \to rchild, x$);
}
```

Algorithm 2.4 Recursive search of a binary search tree

If we wish to search by rank, each node should have an additional field *leftsize*, which is one plus the number of elements in the left subtree of the node. For the search tree of Figure 2.11(b), the nodes with keys 2, 5, 30, and 40, respectively, have *leftsize* equal to 1, 2, 3, and 1. Algorithm 2.6 searches for the kth-smallest element.

As can be seen, a binary search tree of height h can be searched by key as well as by rank in $O(h)$ time.

```
Algorithm ISearch($x$)
{
    $found$ := false;
    $t$ := $tree$;
    while (($t \neq 0$) and not $found$) do
    {
        if ($x = (t \rightarrow data)$) then $found$ := true;
        else if ($x < (t \rightarrow data)$) then $t := (t \rightarrow lchild)$;
            else $t := (t \rightarrow rchild)$;
    }
    if (not $found$) then return 0;
    else return $t$;
}
```

Algorithm 2.5 Iterative search of a binary search tree

```
Algorithm Searchk($k$)
{
    $found$ := false; $t$ := $tree$;
    while (($t \neq 0$) and not $found$) do
    {
        if ($k = (t \rightarrow leftsize)$) then $found$ := true;
        else if ($k < (t \rightarrow leftsize)$) then $t := (t \rightarrow lchild)$;
            else
            {
                $k := k - (t \rightarrow leftsize)$;
                $t := (t \rightarrow rchild)$;
            }
    }
    if (not $found$) then return 0;
    else return $t$;
}
```

Algorithm 2.6 Searching a binary search tree by rank

Insertion into a Binary Search Tree

To insert a new element x, we must first verify that its key is different from those of existing elements. To do this, a search is carried out. If the search is unsuccessful, then the element is inserted at the point the search terminated. For instance, to insert an element with key 80 into the tree of Figure 2.12(a), we first search for 80. This search terminates unsuccessfully, and the last node examined is the one with key 40. The new element is inserted as the right child of this node. The resulting search tree is shown in Figure 2.12(b). Figure 2.12(c) shows the result of inserting the key 35 into the search tree of Figure 2.12(b).

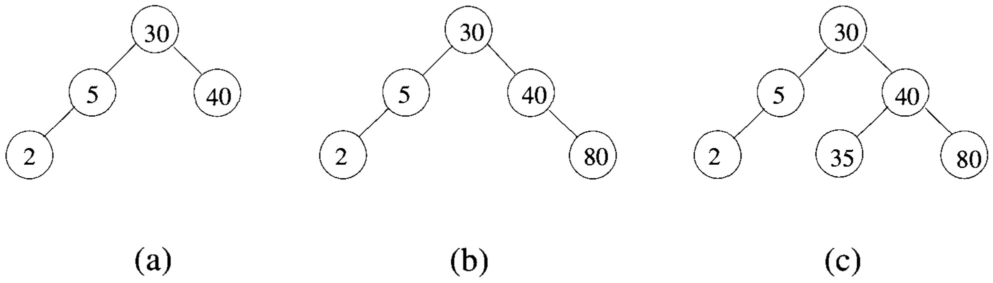


Figure 2.12 Inserting into a binary search tree

Algorithm 2.7 implements the insert strategy just described. If a node has a *leftsize* field, then this is updated too. Regardless, the insertion can be performed in $O(h)$ time, where h is the height of the search tree.

Deletion from a Binary Search Tree

Deletion of a leaf element is quite easy. For example, to delete 35 from the tree of Figure 2.12(c), the left-child field of its parent is set to 0 and the node disposed. This gives us the tree of Figure 2.12(b). To delete the 80 from this tree, the right-child field of 40 is set to 0; this gives the tree of Figure 2.12(a). Then the node containing 80 is disposed.

The deletion of a nonleaf element that has only one child is also easy. The node containing the element to be deleted is disposed, and the single child takes the place of the disposed node. So, to delete the element 5 from the tree of Figure 2.12(b), we simply change the pointer from the parent node (i.e., the node containing 30) to the single-child node (i.e., the node containing 2).

```
Algorithm Insert($x$)
// Insert $x$ into the binary search tree.
{
    $found$ := false;
    $p$ := $tree$;
    // Search for $x$. $q$ is the parent of $p$.
    while (($p \neq 0$) and not $found$) do
    {
        $q := p$; // Save $p$.
        if ($x = (p \rightarrow data)$) then $found$ := true;
        else  if ($x < (p \rightarrow data)$) then $p := (p \rightarrow lchild)$;
              else $p := (p \rightarrow rchild)$;
    }

    // Perform insertion.
    if (not $found$) then
    {
        $p$ := new $TreeNode$;
        $(p \rightarrow lchild) := 0$; $(p \rightarrow rchild) := 0$; $(p \rightarrow data) := x$;
        if ($tree \neq 0$) then
        {
            if ($x < (q \rightarrow data)$) then $(q \rightarrow lchild) := p$;
            else $(q \rightarrow rchild) := p$;
        }
        else $tree := p$;
    }
}
```

Algorithm 2.7 Insertion into a binary search tree

When the element to be deleted is in a nonleaf node that has two children, the element is replaced by either the largest element in its left subtree or the smallest one in its right subtree. Then we proceed to delete this replacing element from the subtree from which it was taken. For instance, if we wish to delete the element with key 30 from the tree of Figure 2.13(a), then we replace it by either the largest element, 5, in its left subtree or the smallest element, 40, in its right subtree. Suppose we opt for the largest element in the left subtree. The 5 is moved into the root, and the tree of Figure 2.13(b) is obtained. Now we must delete the second 5. Since this node has only one child, the pointer from its parent is changed to point to this child. The tree of Figure 2.13(c) is obtained. We can verify that regardless of whether the replacing element is the largest in the left subtree or the smallest in the right subtree, it is originally in a node with a degree of at most one. So, deleting it from this node is quite easy. We leave the writing of the deletion procedure as an exercise. It should be evident that a deletion can be performed in $O(h)$ time if the search tree has a height of h.

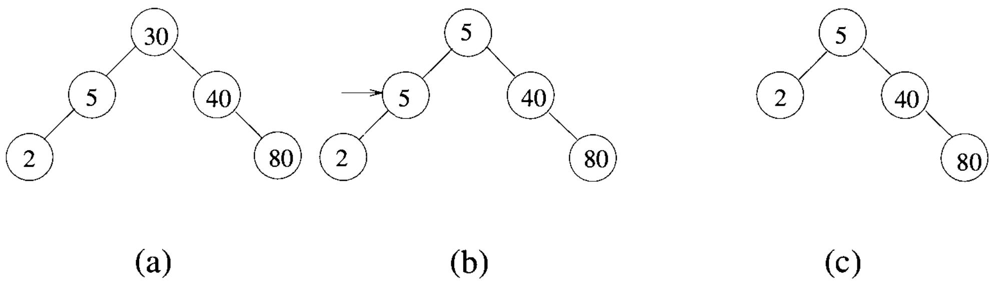


Figure 2.13 Deletion from a binary search tree

Height of a Binary Search Tree

Unless care is taken, the height of a binary search tree with n elements can become as large as n. This is the case, for instance, when Algorithm 2.7 is used to insert the keys $[1, 2, 3, \ldots, n]$, in this order, into an initially empty binary search tree. It can, however, be shown that when insertions and deletions are made at random using the procedures given here, the height of the binary search tree is $O(\log n)$ on the average.

Search trees with a worst-case height of $O(\log n)$ are called *balanced search trees.* Balanced search trees that permit searches, inserts, and deletes to be performed in $O(\log n)$ time are listed in Table 2.1. Examples include AVL trees, 2-3 trees, Red-Black trees, and B-trees. On the other hand splay trees take $O(\log n)$ time for each of these operations in the *amortized* sense. A description of these balanced trees can be found in the book by E. Horowitz, S. Sahni, and D. Mehta cited at the end of this chapter.

Data structure	search	insert	delete
Binary search tree	$O(n)$ (wc) $O(\log n)$ (av)	$O(n)$ (wc) $O(\log n)$ (av)	$O(n)$ (wc) $O(\log n)$ (av)
AVL tree	$O(\log n)$ (wc)	$O(\log n)$ (wc)	$O(\log n)$ (wc)
2-3 tree	$O(\log n)$ (wc)	$O(\log n)$ (wc)	$O(\log n)$ (wc)
Red-Black tree	$O(\log n)$ (wc)	$O(\log n)$ (wc)	$O(\log n)$ (wc)
B-tree	$O(\log n)$ (wc)	$O(\log n)$ (wc)	$O(\log n)$ (wc)
Splay tree	$O(\log n)$ (am)	$O(\log n)$ (am)	$O(\log n)$ (am)

Table 2.1 Summary of dictionary implementations. Here (wc) stands for worst case, (av) for average case, and (am) for amortized cost.

2.3.2 Cost Amortization

Suppose that a sequence I1, I2, D1, I3, I4, I5, I6, D2, I7 of insert and delete operations is performed on a set. Assume that the *actual cost* of each of the seven inserts is one. (We use the terms *cost* and *complexity* interchangeably.) By this, we mean that each insert takes one unit of time. Further, suppose that the delete operations D1 and D2 have an actual cost of eight and ten, respectively. So, the total cost of the sequence of operations is 25.

In an amortization scheme we charge some of the actual cost of an operation to other operations. This reduces the charged cost of some operations and increases that of others. The *amortized cost* of an operation is the total cost charged to it. The cost transferring (amortization) scheme is required to be such that the sum of the amortized costs of the operations is greater than or equal to the sum of their actual costs. If we charge one unit of the cost of a delete operation to each of the inserts since the last delete operation (if any), then two units of the cost of D1 get transferred to I1 and I2 (the charged cost of each increases by one), and four units of the cost of D2 get transferred to I3 to I6. The amortized cost of each of I1 to I6 becomes two, that of I7 becomes equal to its actual cost (that is, one), and that of each of D1 and D2 becomes 6. The sum of the amortized costs is 25, which is the same as the sum of the actual costs.

Now suppose we can prove that no matter what sequence of insert and delete operations is performed, we can charge costs in such a way that the amortized cost of each insertion is no more than two and that of each deletion is no more than six. This enables us to claim that the actual cost of any insert/delete sequence is no more than $2 * i + 6 * d$, where i and d are, respectively, the number of insert and delete operations in the sequence. Suppose that the actual cost of a deletion is no more than ten and that of an insertion is one. Using actual costs, we can conclude that the sequence cost is no more than $i + 10 * d$. Combining these two bounds, we obtain $\min\{2 * i + 6 * d,\ i + 10 * d\}$ as a bound on the sequence cost. Hence, using the notion of cost amortization, we can obtain tighter bounds on the complexity of a sequence of operations.

The amortized time complexity to perform insert, delete, and search operations in splay trees is $O(\log n)$. This amortization is over n operations. In other words, the total time taken for processing an arbitrary sequence of n operations is $O(n \log n)$. Some operations may take much longer than $O(\log n)$ time, but when amortized over n operations, each operation costs $O(\log n)$ time.

EXERCISES

1. Write an algorithm to delete an element x from a binary search tree t. What is the time complexity of your algorithm?

2. Present an algorithm to start with an initially empty binary search tree and make n random insertions. Use a uniform random number generator to obtain the values to be inserted. Measure the height of the resulting binary search tree and divide this height by $\log_2 n$. Do this for $n = 100, 500, 1,000, 2,000, 3,000, \ldots, 10,000$. Plot the ratio height$/\log_2 n$ as a function of n. The ratio should be approximately constant (around 2). Verify that this is so.

3. Suppose that each node in a binary search tree also has the field *leftsize* as described in the text. Design an algorithm to insert an element x into such a binary search tree. The complexity of your algorithm should be $O(h)$, where h is the height of the search tree. Show that this is the case.

4. Do Exercise 3, but this time present an algorithm to delete the element with the kth-smallest key in the binary search tree.

5. Find an efficient data structure for representing a subset S of the integers from 1 to n. Operations we wish to perform on the set are

   - **INSERT**(i) to insert the integer i to the set S. If i is already in the set, this instruction must be ignored.
   - **DELETE** to delete an arbitrary member from the set.
   - **MEMBER**(i) to check whether i is a member of the set.

Your data structure should enable each one of the above operations in constant time (irrespective of the cardinality of S).

6. Any algorithm that merges two sorted lists of size n and m, respectively, must make at least $n + m - 1$ comparisons in the worst case. What implications does this have on the run time of any comparison-based algorithm that combines two binary search trees that have n and m elements, respectively?

7. It is known that every comparison-based algorithm to sort n elements must make $O(n \log n)$ comparisons in the worst case. What implications does this have on the complexity of initializing a binary search tree with n elements?

2.4 PRIORITY QUEUES

Any data structure that supports the operations of search min (or max), insert, and delete min (or max, respectively) is called a *priority queue.*

Example 2.2 Suppose that we are selling the services of a machine. Each user pays a fixed amount per use. However, the time needed by each user is different. We wish to maximize the returns from this machine under the assumption that the machine is not to be kept idle unless no user is available. This can be done by maintaining a priority queue of all persons waiting to use the machine. Whenever the machine becomes available, the user with the smallest time requirement is selected. Hence, a priority queue that supports delete min is required. When a new user requests the machine, his or her request is put into the priority queue.

If each user needs the same amount of time on the machine but people are willing to pay different amounts for the service, then a priority queue based on the amount of payment can be maintained. Whenever the machine becomes available, the user willing to pay the most is selected. This requires a delete max operation. □

Example 2.3 Suppose that we are simulating a large factory. This factory has many machines and many jobs that require processing on some of the machines. An *event* is said to occur whenever a machine completes the processing of a job. When an event occurs, the job has to be moved to the queue for the next machine (if any) that it needs. If this queue is empty, the job can be assigned to the machine immediately. Also, a new job can be scheduled on the machine that has become idle (provided that its queue is not empty).

To determine the occurrence of events, a priority queue is used. This queue contains the finish time of all jobs that are presently being worked on.

The next event occurs at the least time in the priority queue. So, a priority queue that supports delete min can be used in this application. □

The simplest way to represent a priority queue is as an unordered linear list. Suppose that we have n elements in this queue and the delete max operation is to be supported. If the list is represented sequentially, additions are most easily performed at the end of this list. Hence, the insert time is $\Theta(1)$. A deletion requires a search for the element with the largest key, followed by its deletion. Since it takes $\Theta(n)$ time to find the largest element in an n-element unordered list, the delete time is $\Theta(n)$. Each deletion takes $\Theta(n)$ time. An alternative is to use an ordered linear list. The elements are in nondecreasing order if a sequential representation is used. The delete time for each representation is $\Theta(1)$ and the insert time $O(n)$. When a *max heap* is used, both additions and deletions can be performed in $O(\log n)$ time.

2.4.1 Heaps

Definition 2.4 [Heap] A *max (min) heap* is a complete binary tree with the property that the value at each node is at least as large as (as small as) the values at its children (if they exist). Call this property the *heap property.* □

In this section we study in detail an efficient way of realizing a priority queue. We might first consider using a queue since inserting new elements would be very efficient. But finding the largest element would necessitate a scan of the entire queue. A second suggestion might be to use a sorted list that is stored sequentially. But an insertion could require moving all of the items in the list. What we want is a data structure that allows *both* operations to be done efficiently. One such data structure is the max heap.

The definition of a max heap implies that one of the largest elements is at the root of the heap. If the elements are distinct, then the root contains the largest item. A max heap can be implemented using an array $a[\ ]$.

To insert an element into the heap, one adds it "at the bottom" of the heap and then compares it with its parent, grandparent, greatgrandparent, and so on, until it is less than or equal to one of these values. Algorithm Insert (Algorithm 2.8) describes this process in detail.

Figure 2.14 shows one example of how Insert would insert a new value into an existing heap of six elements. It is clear from Algorithm 2.8 and Figure 2.14 that the time for Insert can vary. In the best case the new element is correctly positioned initially and no values need to be rearranged. In the worst case the number of executions of the **while** loop is proportional to the number of levels in the heap. Thus if there are n elements in the heap, inserting a new element takes $\Theta(\log n)$ time in the worst case.

```
Algorithm Insert($a, n$)
{
    // Inserts $a[n]$ into the heap which is stored in $a[1 : n - 1]$.
    $i := n$; $item := a[n]$;
    while (($i > 1$) and ($a[\lfloor i/2 \rfloor] < item$)) do
    {
        $a[i] := a[\lfloor i/2 \rfloor]$; $i := \lfloor i/2 \rfloor$;
    }
    $a[i] := item$; return true;
}
```

Algorithm 2.8 Insertion into a heap

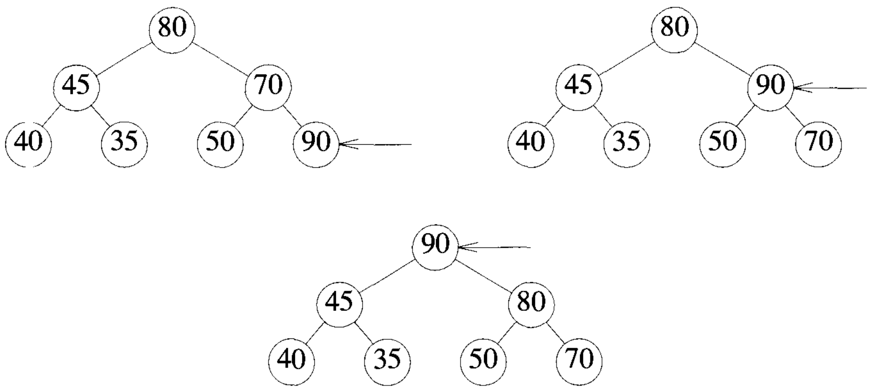


Figure 2.14 Action of Insert inserting 90 into an existing heap

To delete the maximum key from the max heap, we use an algorithm called Adjust. Adjust takes as input the array $a[\ ]$ and the integers i and n. It regards $a[1:n]$ as a complete binary tree. If the subtrees rooted at $2i$ and $2i+1$ are already max heaps, then Adjust will rearrange elements of $a[\ ]$ such that the tree rooted at i is also a max heap. The maximum element from the max heap $a[1:n]$ can be deleted by deleting the root of the corresponding complete binary tree. The last element of the array, that is, $a[n]$, is copied to the root, and finally we call Adjust($a, 1, n-1$). Both Adjust and DelMax are described in Algorithm 2.9.

```
Algorithm Adjust($a, i, n$)
// The complete binary trees with roots $2i$ and $2i+1$ are
// combined with node $i$ to form a heap rooted at $i$. No
// node has an address greater than $n$ or less than 1.
{
    $j := 2i$; $item := a[i]$;
    while ($j \leq n$) do
    {
        if (($j < n$) and ($a[j] < a[j+1]$)) then $j := j+1$;
            // Compare left and right child
            // and let $j$ be the larger child.
        if ($item \geq a[j]$) then break;
            // A position for $item$ is found.
        $a[\lfloor j/2 \rfloor] := a[j]$; $j := 2j$;
    }
    $a[\lfloor j/2 \rfloor] := item$;
}
```

```
Algorithm DelMax($a, n, x$)
// Delete the maximum from the heap $a[1:n]$ and store it in $x$.
{
    if ($n = 0$) then
    {
        write ("heap is empty"); return false;
    }
    $x := a[1]$; $a[1] := a[n]$;
    Adjust($a, 1, n-1$); return true;
}
```

Algorithm 2.9 Deletion from a heap

Note that the worst-case run time of Adjust is also proportional to the height of the tree. Therefore, if there are n elements in a heap, deleting the maximum can be done in $O(\log n)$ time.

To sort n elements, it suffices to make n insertions followed by n deletions from a heap. Algorithm 2.10 has the details. Since insertion and deletion take $O(\log n)$ time each in the worst case, this sorting algorithm has a time complexity of $O(n \log n)$.

```
Algorithm Sort($a, n$)
// Sort the elements $a[1 : n]$.
{
    for $i := 1$ to $n$ do Insert($a, i$);
    for $i := n$ to 1  step $-1$ do
    {
        DelMax($a, i, x$); $a[i] := x$;
    }
}
```

Algorithm 2.10 A sorting algorithm

It turns out that we can insert n elements into a heap faster than we can apply Insert n times. Before getting into the details of the new algorithm, let us consider how the n inserts take place. Figure 2.15 shows how the data (40, 80, 35, 90, 45, 50, and 70) move around until a heap is created when using Insert. Trees to the left of any $\rightarrow$ represent the state of the array $a[1 : i]$ before some call of Insert. Trees to the right of $\rightarrow$ show how the array was altered by Insert to produce a heap. The array is drawn as a complete binary tree for clarity.

The data set that causes the heap creation method using Insert to behave in the worst way is a set of elements in ascending order. Each new element rises to become the new root. There are at most 2^{i-1} nodes on level i of a complete binary tree, $1 \leq i \leq \lceil \log_2(n+1) \rceil$. For a node on level i the distance to the root is $i - 1$. Thus the worst-case time for heap creation using Insert is

$$\sum_{1 \leq i \leq \lceil \log_2(n+1) \rceil} (i-1)2^{i-1} < \lceil \log_2(n+1) \rceil 2^{\lceil \log_2(n+1) \rceil} = O(n \log n)$$

A surprising fact about Insert is that its average behavior on n random inputs is asymptotically faster than its worst case, $O(n)$ rather than $O(n \log n)$.

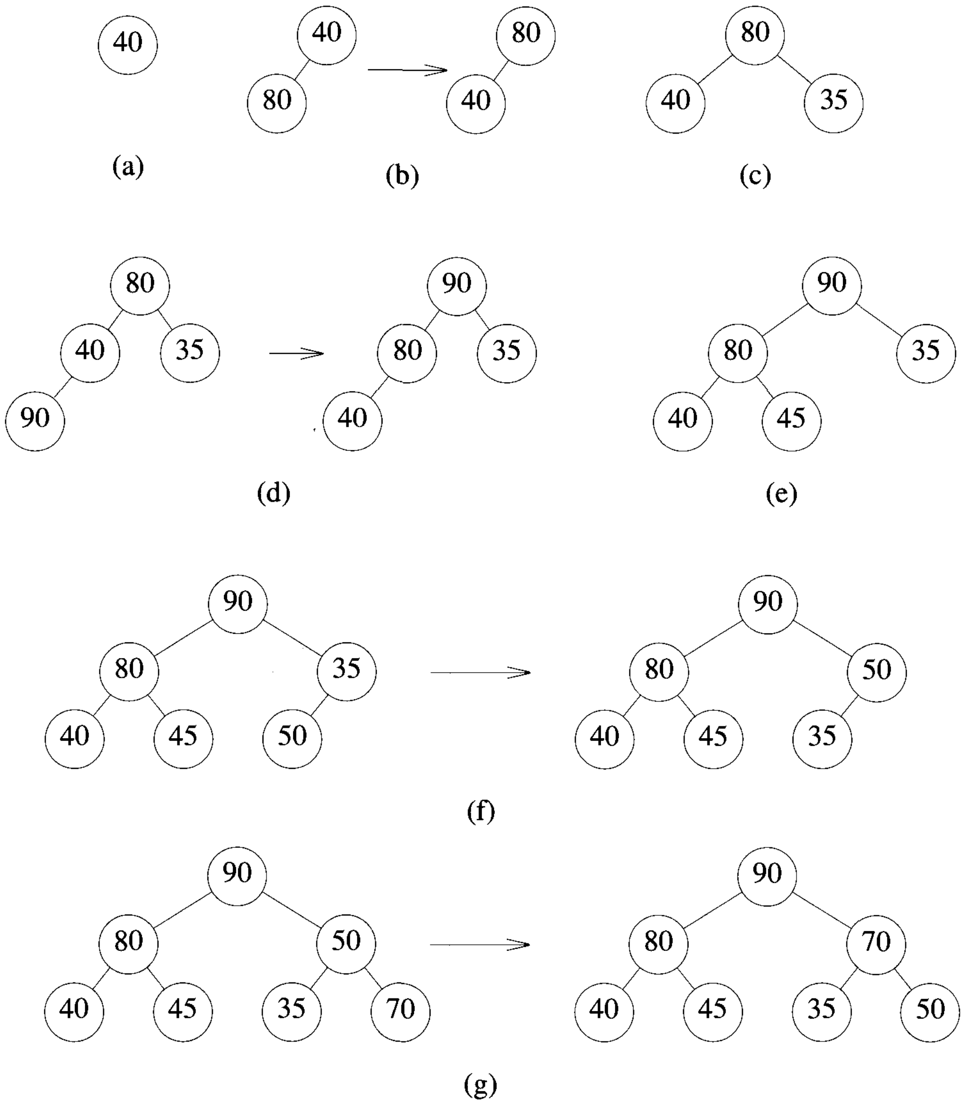


Figure 2.15 Forming a heap from the set $\{40, 80, 35, 90, 45, 50, 70\}$

This implies that on the average each new value only rises a constant number of levels in the tree.

It is possible to devise an algorithm that can perform n inserts in $O(n)$ time rather than $O(n \log n)$. This reduction is achieved by an algorithm that regards any array $a[1:n]$ as a complete binary tree and works from the leaves up to the root, level by level. At each level, the left and right subtrees of any node are heaps. Only the value in the root node may violate the heap property.

Given n elements in $a[1:n]$, we can create a heap by applying Adjust. It is easy to see that leaf nodes are already heaps. So we can begin by calling Adjust for the parents of leaf nodes and then work our way up, level by level, until the root is reached. The resultant algorithm is Heapify (Algorithm 2.11). In Figure 2.16 we observe the action of Heapify as it creates a heap out of the given seven elements. The initial tree is drawn in Figure 2.16(a). Since $n = 7$, the first call to Adjust has $i = 3$. In Figure 2.16(b) the three elements 118, 151, and 132 are rearranged to form a heap. Subsequently Adjust is called with $i = 2$ and $i = 1$; this gives the trees in Figure 2.16(c) and (d).

```
Algorithm Heapify($a, n$)
// Readjust the elements in $a[1:n]$ to form a heap.
{
    for $i := \lfloor n/2 \rfloor$ to 1 step $-1$ do Adjust($a, i, n$);
}
```

Algorithm 2.11 Creating a heap out of n arbitrary elements

For the worst-case analysis of Heapify let $2^{k-1} \leq n < 2^k$, where $k = \lceil \log(n+1) \rceil$, and recall that the levels of the n-node complete binary tree are numbered 1 to k. The worst-case number of iterations for Adjust is $k - i$ for a node on level i. The total time for Heapify is proportional to

$$\sum_{1 \leq i \leq k} 2^{i-1}(k-i) = \sum_{1 \leq i \leq k-1} i2^{k-i-1} \leq n \sum_{1 \leq i \leq k-1} i/2^i \leq 2n = O(n) \qquad (2.1)$$

Comparing Heapify with the repeated use of Insert, we see that the former is faster in the worst case, requiring $O(n)$ versus $O(n \log n)$ operations. However, Heapify requires that all the elements be available before heap creation begins. Using Insert, we can add a new element into the heap at any time.

Our discussion on insert, delete, and so on, so far has been with respect to a max heap. It should be easy to see that a parallel discussion could have

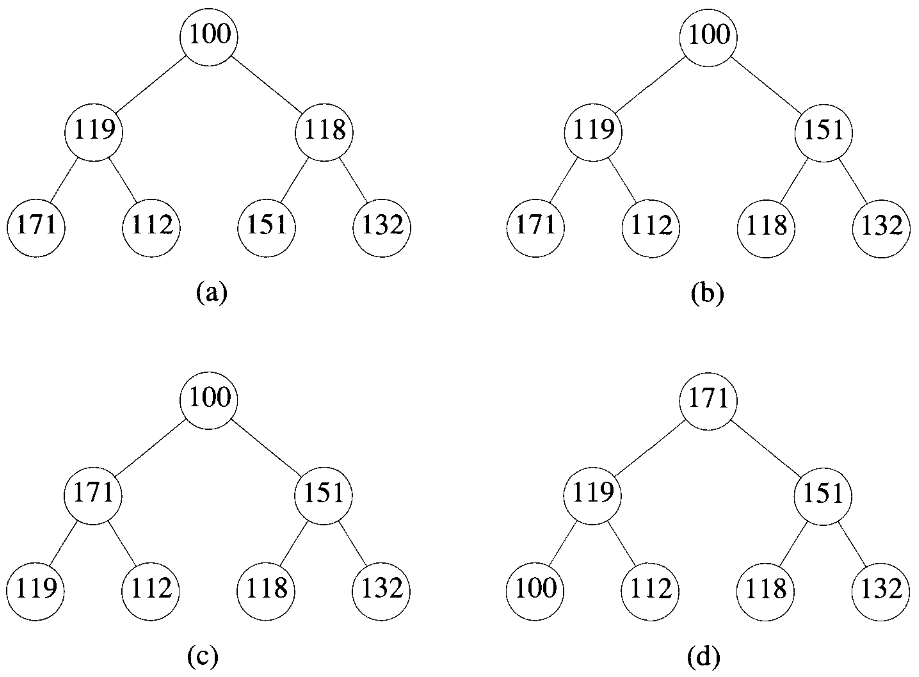


Figure 2.16 Action of Heapify$(a, 7)$ on the data (100, 119, 118, 171, 112, 151, and 132)

been carried out with respect to a min heap. For a min heap it is possible to delete the smallest element in $O(\log n)$ time and also to insert an element in $O(\log n)$ time.

2.4.2 Heapsort

The best-known example of the use of a heap arises in its application to sorting. A conceptually simple sorting strategy has been given before, in which the maximum value is continually removed from the remaining unsorted elements. A sorting algorithm that incorporates the fact that n elements can be inserted in $O(n)$ time is given in Algorithm 2.12.

```
Algorithm HeapSort($a, n$)
// $a[1:n]$ contains n elements to be sorted. HeapSort
// rearranges them inplace into nondecreasing order.
{
    Heapify($a, n$); // Transform the array into a heap.
    // Interchange the new maximum with the element
    // at the end of the array.
    for $i := n$ to 2 step $-1$ do
    {
        $t := a[i]; a[i] := a[1]; a[1] := t;$
        Adjust($a, 1, i-1$);
    }
}
```

Algorithm 2.12 Heapsort

Though the call of Heapify requires only $O(n)$ operations, Adjust possibly requires $O(\log n)$ operations for each invocation. Thus the worst-case time is $O(n \log n)$. Notice that the storage requirements, besides $a[1:n]$, are only for a few simple variables.

A number of other data structures can also be used to implement a priority queue. Examples include the binomial heap, deap, Fibonacci heap, and so on. A description of these can be found in the book by E. Horowitz, S. Sahni, and D. Mehta. Table 2.2 summarizes the performances of these data structures. Many of these data structures support the operations of deleting and searching for arbitrary elements (Red-Black tree being an example), in addition to the ones needed for a priority queue.

Data structure	insert	delete min
Min heap	$O(\log n)$ (wc)	$O(\log n)$ (wc)
Min-max heap	$O(\log n)$ (wc)	$O(\log n)$ (wc)
Deap	$O(\log n)$ (wc)	$O(\log n)$ (wc)
Leftist tree	$O(\log n)$ (wc)	$O(\log n)$ (wc)
Binomial heap	$O(\log n)$ (wc) $O(1)$ (am)	$O(\log n)$ (wc) $O(\log n)$ (am)
Fibonacci heap	$O(\log n)$ (wc) $O(1)$ (am)	$O(\log n)$ (wc) $O(\log n)$ (am)
2-3 tree	$O(\log n)$ (wc)	$O(\log n)$ (wc)
Red-Black tree	$O(\log n)$ (wc)	$O(\log n)$ (wc)

Table 2.2 Performances of different data structures when realizing a priority queue. Here (wc) stands for worst case and (am) denotes amortized cost.

EXERCISES

1. Verify for yourself that algorithm Insert (Algorithm 2.8) uses only a constant number of comparisons to insert a random element into a heap by performing an appropriate experiment.

2. (a) Equation 2.1 makes use of the fact that the sum $\sum_{i=1}^{\infty} \frac{i}{2^i}$ converges. Prove this fact.

   (b) Use induction to show that $\sum_{i=1}^{k} 2^{i-1}(k-i) = 2^k - k - 1, k \geq 1$.

3. Program and run algorithm HeapSort (Algorithm 2.12) and compare its time with the time of any of the sorting algorithms discussed in Chapter 1.

4. Design a data structure that supports the following operations: INSERT and MIN. The worst-case run time should be $O(1)$ for each of these operations.

5. Notice that a binary search tree can be used to implement a priority queue.

   (a) Present an algorithm to delete the largest element in a binary search tree. Your procedure should have complexity $O(h)$, where h is the height of the search tree. Since h is $O(\log n)$ on average, you can perform each of the priority queue operations in average time $O(\log n)$.

(b) Compare the performances of max heaps and binary search trees as data structures for priority queues. For this comparison, generate random sequences of insert and delete max operations and measure the total time taken for each sequence by each of these data structures.

6. Input is a sequence X of n keys with many duplications such that the number of distinct keys is d ($< n$). Present an $O(n \log d)$-time sorting algorithm for this input. (For example, if $X = 5, 6, 1, 18, 6, 4, 4, 1, 5, 17$, the number of distinct keys in X is six.)

2.5 SETS AND DISJOINT SET UNION

2.5.1 Introduction

In this section we study the use of forests in the representation of sets. We shall assume that the elements of the sets are the numbers $1, 2, 3, \ldots, n$. These numbers might, in practice, be indices into a symbol table in which the names of the elements are stored. We assume that the sets being represented are pairwise disjoint (that is, if S_i and S_j, $i \neq j$, are two sets, then there is no element that is in both S_i and S_j). For example, when $n = 10$, the elements can be partitioned into three disjoint sets, $S_1 = \{1, 7, 8, 9\}$, $S_2 = \{2, 5, 10\}$, and $S_3 = \{3, 4, 6\}$. Figure 2.17 shows one possible representation for these sets. In this representation, each set is represented as a tree. Notice that for each set we have linked the nodes from the children to the parent, rather than our usual method of linking from the parent to the children. The reason for this change in linkage becomes apparent when we discuss the implementation of set operations.

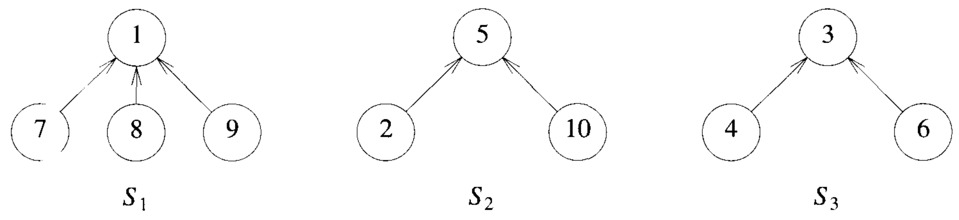


Figure 2.17 Possible tree representation of sets

The operations we wish to perform on these sets are:

1. **Disjoint set union**. If S_i and S_j are two disjoint sets, then their union $S_i \cup S_j$ = all elements x such that x is in S_i or S_j. Thus, $S_1 \cup S_2 = \{1, 7, 8, 9, 2, 5, 10\}$. Since we have assumed that all sets are disjoint, we can assume that following the union of S_i and S_j, the sets S_i and S_j do not exist independently; that is, they are replaced by $S_i \cup S_j$ in the collection of sets.

2. **Find**(i). Given the element i, find the set containing i. Thus, 4 is in set S_3, and 9 is in set S_1.

2.5.2 Union and Find Operations

Let us consider the union operation first. Suppose that we wish to obtain the union of S_1 and S_2 (from Figure 2.17). Since we have linked the nodes from children to parent, we simply make one of the trees a subtree of the other. $S_1 \cup S_2$ could then have one of the representations of Figure 2.18.

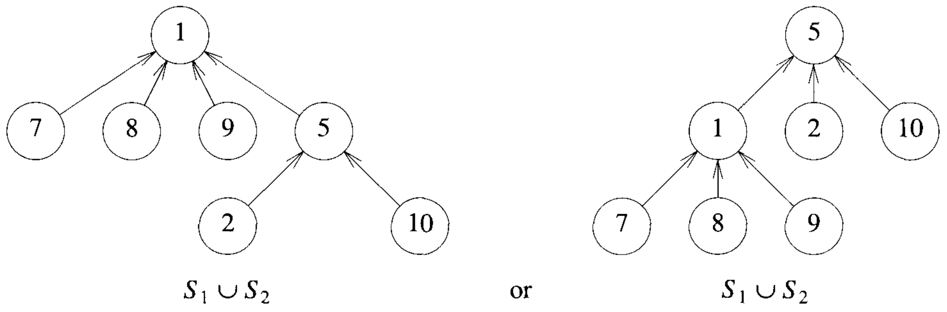


Figure 2.18 Possible representations of $S_1 \cup S_2$

To obtain the union of two sets, all that has to be done is to set the parent field of one of the roots to the other root. This can be accomplished easily if, with each set name, we keep a pointer to the root of the tree representing that set. If, in addition, each root has a pointer to the set name, then to determine which set an element is currently in, we follow parent links to the root of its tree and use the pointer to the set name. The data representation for S_1, S_2, and S_3 may then take the form shown in Figure 2.19.

In presenting the union and find algorithms, we ignore the set names and identify sets just by the roots of the trees representing them. This simplifies

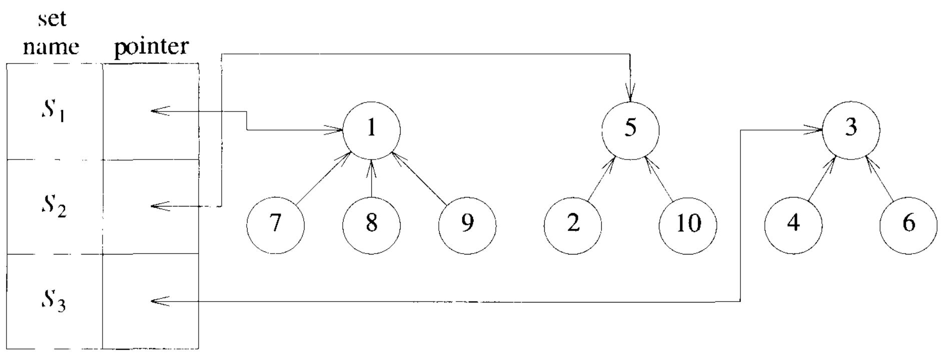


Figure 2.19 Data representation for S_1, S_2, and S_3

the discussion. The transition to set names is easy. If we determine that element i is in a tree with root j, and j has a pointer to entry k in the set name table, then the set name is just $name[k]$. If we wish to unite sets S_i and S_j, then we wish to unite the trees with roots FindPointer(S_i) and FindPointer(S_j). Here FindPointer is a function that takes a set name and determines the root of the tree that represents it. This is done by an examination of the [set name, pointer] table. In many applications the set name is just the element at the root. The operation of $Find(i)$ now becomes: Determine the root of the tree containing element i. The function $Union(i, j)$ requires two trees with roots i and j be joined. Also to simplify, assume that the set elements are the numbers 1 through n.

Since the set elements are numbered 1 through n, we represent the tree nodes using an array $p[1 : n]$, where n is the maximum number of elements. The ith element of this array represents the tree node that contains element i. This array element gives the parent pointer of the corresponding tree node. Figure 2.20 shows this representation of the sets S_1, S_2, and S_3 of Figure 2.17. Notice that root nodes have a parent of -1.

i	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
p	-1	5	-1	3	-1	3	1	1	1	5

Figure 2.20 Array representation of S_1, S_2, and S_3 of Figure 2.17

We can now implement *Find*(i) by following the indices, starting at i until we reach a node with parent value -1. For example, *Find*(6) starts at 6 and then moves to 6's parent, 3. Since $p[3]$ is negative, we have reached the root. The operation *Union*(i, j) is equally simple. We pass in two trees with roots i and j. Adopting the convention that the first tree becomes a subtree of the second, the statement $p[i] := j$; accomplishes the union.

```
Algorithm SimpleUnion($i, j$)
{
    $p[i] := j$;
}

Algorithm SimpleFind($i$)
{
    while ($p[i] \geq 0$) do $i := p[i]$;
    return $i$;
}
```

Algorithm 2.13 Simple algorithms for union and find

Algorithm 2.13 gives the descriptions of the union and find operations just discussed. Although these two algorithms are very easy to state, their performance characteristics are not very good. For instance, if we start with q elements each in a set of its own (that is, $S_i = \{i\}$, $1 \leq i \leq q$), then the initial configuration consists of a forest with q nodes, and $p[i] = 0$, $1 \leq i \leq q$. Now let us process the following sequence of *union-find* operations:

$$Union(1,2),\ Union(2,3),\ Union(3,4),\ Union(4,5), \ldots,\ Union(n-1,n)$$

$$Find(1),\ Find(2), \ldots,\ Find(n)$$

This sequence results in the degenerate tree of Figure 2.21.

Since the time taken for a union is constant, the $n - 1$ unions can be processed in time $O(n)$. However, each find requires following a sequence of *parent* pointers from the element to be found to the root. Since the time required to process a find for an element at level i of a tree is $O(i)$, the total time needed to process the n finds is $O(\sum_{i=1}^{n} i) = O(n^2)$.

We can improve the performance of our union and find algorithms by avoiding the creation of degenerate trees. To accomplish this, we make use of a weighting rule for *Union*(i, j).

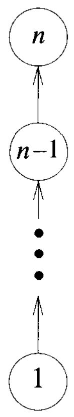


Figure 2.21 Degenerate tree

Definition 2.5 [Weighting rule for *Union(i,j)*] If the number of nodes in the tree with root i is less than the number in the tree with root j, then make j the parent of i; otherwise make i the parent of j. □

When we use the weighting rule to perform the sequence of set unions given before, we obtain the trees of Figure 2.22. In this figure, the unions have been modified so that the input parameter values correspond to the roots of the trees to be combined.

To implement the weighting rule, we need to know how many nodes there are in every tree. To do this easily, we maintain a *count* field in the root of every tree. If i is a root node, then $count[i]$ equals the number of nodes in that tree. Since all nodes other than the roots of trees have a positive number in the p field, we can maintain the count in the p field of the roots as a negative number.

Using this convention, we obtain Algorithm 2.14. In this algorithm the time required to perform a union has increased somewhat but is still bounded by a constant (that is, it is $O(1)$). The find algorithm remains unchanged. The maximum time to perform a find is determined by Lemma 2.3.

Lemma 2.3 Assume that we start with a forest of trees, each having one node. Let T be a tree with m nodes created as a result of a sequence of unions each performed using WeightedUnion. The height of T is no greater than $\lfloor \log_2 m \rfloor + 1$.

Proof: The lemma is clearly true for $m = 1$. Assume it is true for all trees with i nodes, $i \leq m - 1$. We show that it is also true for $i = m$.

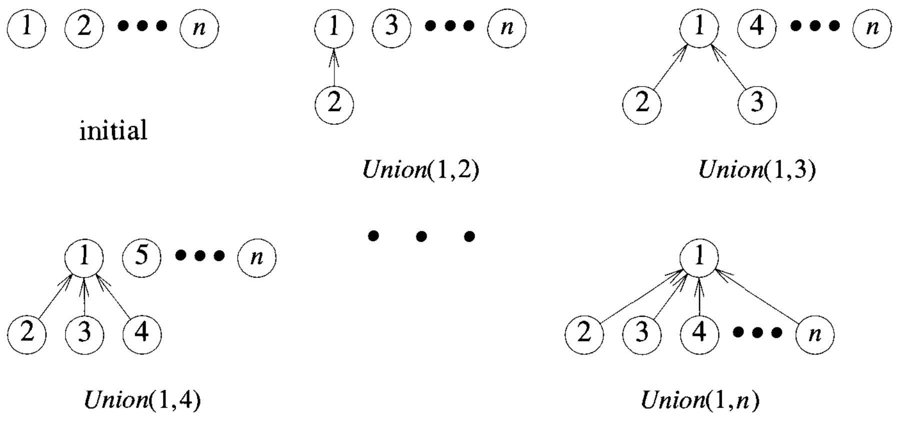


Figure 2.22 Trees obtained using the weighting rule

```
Algorithm WeightedUnion$(i, j)$
// Union sets with roots $i$ and $j$, $i \neq j$, using the
// weighting rule. $p[i] = -count[i]$ and $p[j] = -count[j]$.
{
    $temp := p[i] + p[j]$;
    if $(p[i] > p[j])$ then
    { // $i$ has fewer nodes.
        $p[i] := j$; $p[j] := temp$;
    }
    else
    { // $j$ has fewer or equal nodes.
        $p[j] := i$; $p[i] := temp$;
    }
}
```

Algorithm 2.14 Union algorithm with weighting rule

Let T be a tree with m nodes created by WeightedUnion. Consider the last union operation performed, $Union(k, j)$. Let a be the number of nodes in tree j, and $m - a$ the number in k. Without loss of generality we can assume $1 \leq a \leq \frac{m}{2}$. Then the height of T is either the same as that of k or is one more than that of j. If the former is the case, the height of T is $\leq \lfloor \log_2(m - a) \rfloor + 1 \leq \lfloor \log_2 m \rfloor + 1$. However, if the latter is the case, the height of T is $\leq \lfloor \log_2 a \rfloor + 2 \leq \lfloor \log_2 \frac{m}{2} \rfloor + 2 \leq \lfloor \log_2 m \rfloor + 1$. □

Example 2.4 shows that the bound of Lemma 2.3 is achievable for some sequence of unions.

Example 2.4 Consider the behavior of WeightedUnion on the following sequence of unions starting from the initial configuration $p[i] = -count[i] = -1$, $1 \leq i \leq 8 = n$:

$$Union(1, 2),\ Union(3, 4),\ Union(5, 6),\ Union(7, 8),$$
$$Union(1, 3),\ Union(5, 7),\ Union(1, 5)$$

The trees of Figure 2.23 are obtained. As is evident, the height of each tree with m nodes is $\lfloor \log_2 m \rfloor + 1$. □

From Lemma 2.3, it follows that the time to process a find is $O(\log m)$ if there are m elements in a tree. If an intermixed sequence of $u - 1$ union and f find operations is to be processed, the time becomes $O(u + f \log u)$, as no tree has more than u nodes in it. Of course, we need $O(n)$ additional time to initialize the n-tree forest.

Surprisingly, further improvement is possible. This time the modification is made in the find algorithm using the *collapsing rule*.

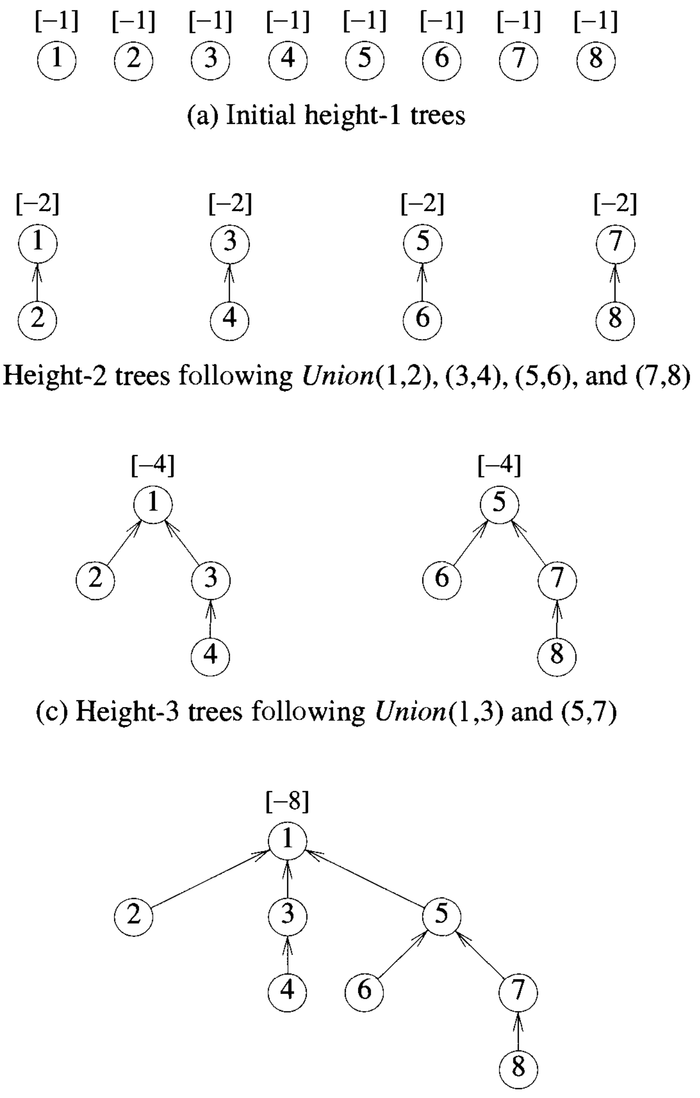
Definition 2.6 [Collapsing rule]: If j is a node on the path from i to its root and $p[i] \neq root[i]$, then set $p[j]$ to $root[i]$. □

CollapsingFind (Algorithm 2.15) incorporates the collapsing rule.

Example 2.5 Consider the tree created by WeightedUnion on the sequence of unions of Example 2.4. Now process the following eight finds:

$$Find(8),\ Find(8), \ldots,\ Find(8)$$

If SimpleFind is used, each $Find(8)$ requires going up three parent link fields for a total of 24 moves to process all eight finds. When CollapsingFind is used, the first $Find(8)$ requires going up three links and then resetting two links. Note that even though only two parent links need to be reset, CollapsingFind will reset three (the parent of 5 is reset to 1). Each of the remaining seven finds requires going up only one link field. The total cost is now only 13 moves. □



(a) Initial height-1 trees

(b) Height-2 trees following *Union*(1,2), (3,4), (5,6), and (7,8)

(c) Height-3 trees following *Union*(1,3) and (5,7)

(d) Height-4 tree following *Union*(1,5)

Figure 2.23 Trees achieving worst-case bound

```
Algorithm CollapsingFind($i$)
// Find the root of the tree containing element $i$. Use the
// collapsing rule to collapse all nodes from $i$ to the root.
{
    $r := i$;
    while ($p[r] > 0$) do $r := p[r]$; // Find the root.
    while ($i \neq r$) do  // Collapse nodes from $i$ to root $r$.
    {
        $s := p[i]$; $p[i] := r$; $i := s$;
    }
    return $r$;
}
```

Algorithm 2.15 Find algorithm with collapsing rule

In the algorithms WeightedUnion and CollapsingFind, use of the collapsing rule roughly doubles the time for an individual find. However, it reduces the worst-case time over a sequence of finds. The worst-case complexity of processing a sequence of unions and finds using WeightedUnion and CollapsingFind is stated in Lemma 2.4. This lemma makes use of a function $\alpha(p, q)$ that is related to a functional inverse of Ackermann's function $A(i, j)$. These functions are defined as follows:

$$\begin{array}{ll} A(1,j) = 2^j & \text{for } j \geq 1 \\ A(i,1) = A(i-1,2) & \text{for } i \geq 2 \\ A(i,j) = A(i-1, A(i, j-1)) & \text{for } i, j \geq 2 \end{array}$$

$$\alpha(p,q) = \min\{z \geq 1 | A(z, \lfloor \frac{p}{q} \rfloor) > \log_2 q\}, \quad p \geq q \geq 1$$

The function $A(i, j)$ is a very rapidly growing function. Consequently, α grows very slowly as p and q are increased. In fact, since $A(3,1) = 16$, $\alpha(p,q) \leq 3$ for $q < 2^{16} = 65,536$ and $p \geq q$. Since $A(4,1)$ is a very large number and in our application q is the number n of set elements and p is $n + f$ (f is the number of finds), $\alpha(p,q) \leq 4$ for all practical purposes.

Lemma 2.4 [Tarjan and Van Leeuwen] Assume that we start with a forest of trees, each having one node. Let $T(f, u)$ be the maximum time required to process any intermixed sequence of f finds and u unions. Assume that $u \geq \frac{n}{2}$. Then

$$k_1[n + f\alpha(f + n, n)] \leq T(f, u) \leq k_2[n + f\alpha(f + n, n)]$$

for some positive constants k_1 and k_2. □

The requirement that $u \geq \frac{n}{2}$ in Lemma 2.4 is really not significant, as when $u < \frac{n}{2}$, some elements are involved in no union operation. These elements remain in singleton sets throughout the sequence of union and find operations and can be eliminated from consideration, as find operations that involve these can be done in $O(1)$ time each. Even though the function $\alpha(f, u)$ is a very slowly growing function, the complexity of our solution to the set representation problem is not linear in the number of unions and finds. The space requirements are one node for each element.

In the exercises, we explore alternatives to the weight rule and the collapsing rule that preserve the time bounds of Lemma 2.4.

EXERCISES

1. Suppose we start with n sets, each containing a distinct element.
   (a) Show that if u unions are performed, then no set contains more than $u + 1$ elements.
   (b) Show that at most $n - 1$ unions can be performed before the number of sets becomes 1.
   (c) Show that if fewer than $\lceil \frac{n}{2} \rceil$ unions are performed, then at least one set with a single element in it remains.
   (d) Show that if u unions are performed, then at least $\max\{n - 2u, 0\}$ singleton sets remain.
2. Experimentally compare the performance of SimpleUnion and SimpleFind (Algorithm 2.13) with WeightedUnion (Algorithm 2.14) and CollapsingFind (Algorithm 2.15). For this, generate a random sequence of union and find operations.
3. (a) Present an algorithm HeightUnion that uses the *height rule* for union operations instead of the weighting rule. This rule is defined below:

   Definition 2.7 [Height rule] If the height of tree i is less than that of tree j, then make j the parent of i; otherwise make i the parent of j. □

   Your algorithm must run in $O(1)$ time and should maintain the height of each tree as a negative number in the p field of the root.

(b) Show that the height bound of Lemma 2.3 applies to trees constructed using the height rule.

(c) Give an example of a sequence of unions that start with n singleton sets and create trees whose heights equal the upper bounds given in Lemma 2.3. Assume that each union is performed using the height rule.

(d) Experiment with the algorithms WeightedUnion (Algorithm 2.14) and HeightUnion to determine which produces better results when used in conjunction with CollapsingFind (Algorithm 2.15).

4. (a) Write an algorithm SplittingFind that uses *path splitting*, defined below, for the find operations instead of path collapsing.

Definition 2.8 [Path splitting] The parent pointer in each node (except the root and its child) on the path from i to the root is changed to point to the node's grandparent. □

Note that when path splitting is used, a single pass from i to the root suffices. R. Tarjan and J. Van Leeuwen have shown that Lemma 2.4 holds when path splitting is used in conjunction with either the weight or the height rule for unions.

(b) Experiment with CollapsingFind (Algorithm 2.15) and SplittingFind to determine which produces better results when used in conjunction with WeightedUnion (Algorithm 2.14).

5. (a) Design an algorithm HalvingFind that uses *path halving*, defined below, for the find operations instead of path collapsing.

Definition 2.9 [Path halving] In path halving, the parent pointer of every other node (except the root and its child) on the path from i to the root is changed to point to the nodes grandparent. □

Note that path halving, like path splitting (Exercise 4), can be implemented with a single pass from i to the root. However, in path halving, only half as many pointers are changed as in path splitting. Tarjan and Van Leeuwen have shown that Lemma 2.4 holds when path halving is used in conjunction with either the weight or the height rule for unions.

(b) Experiment with CollapsingFind and HalvingFind to determine which produces better results when used in conjunction with WeightedUnion (Algorithm 2.14).

2.6 GRAPHS

2.6.1 Introduction

The first recorded evidence of the use of graphs dates back to 1736, when Leonhard Euler used them to solve the now classical Königsberg bridge problem. In the town of Königsberg (now Kaliningrad) the river Pregel (Pregolya) flows around the island Kneiphof and then divides into two. There are, therefore, four land areas that have this river on their borders (see Figure 2.24(a)). These land areas are interconnected by seven bridges, labeled a to g. The land areas themselves are labeled A to D. The Königsberg bridge problem is to determine whether, starting at one land area, it is possible to walk across all the bridges exactly once in returning to the starting land area. One possible walk: Starting from land area B, walk across bridge a to island A, take bridge e to area D, take bridge g to C, take bridge d to A, take bridge b to B, and take bridge f to D.

This walk does not go across all bridges exactly once, nor does it return to the starting land area B. Euler answered the Königsberg bridge problem in the negative: The people of Königsberg cannot walk across each bridge exactly once and return to the starting point. He solved the problem by representing the land areas as vertices and the bridges as edges in a graph (actually a multigraph) as in Figure 2.24(b). His solution is elegant and applies to all graphs. Defining the *degree* of a vertex to be the number of edges incident to it, Euler showed that there is a walk starting at any vertex, going through each edge exactly once and terminating at the start vertex if and only if the degree of each vertex is even. A walk that does this is called *Eulerian*. There is no Eulerian walk for the Königsberg bridge problem, as all four vertices are of odd degree.

Since this first application, graphs have been used in a wide variety of applications. Some of these applications are the analysis of electric circuits, finding shortest routes, project planning, identification of chemical compounds, statistical mechanics, genetics, cybernetics, linguistics, social sciences, and so on. Indeed, it might well be said that of all mathematical structures, graphs are the most widely used.

2.6.2 Definitions

A graph G consists of two sets V and E. The set V is a finite, nonempty set of *vertices*. The set E is a set of pairs of vertices; these pairs are called *edges*. The notations $V(G)$ and $E(G)$ represent the sets of vertices and edges, respectively, of graph G. We also write $G = (V, E)$ to represent a graph. In an *undirected graph* the pair of vertices representing any edge is unordered. Thus, the pairs (u, v) and (v, u) represent the same edge. In a *directed graph* each edge is represented by a directed pair $\langle u, v\rangle$; u is the *tail* and v the

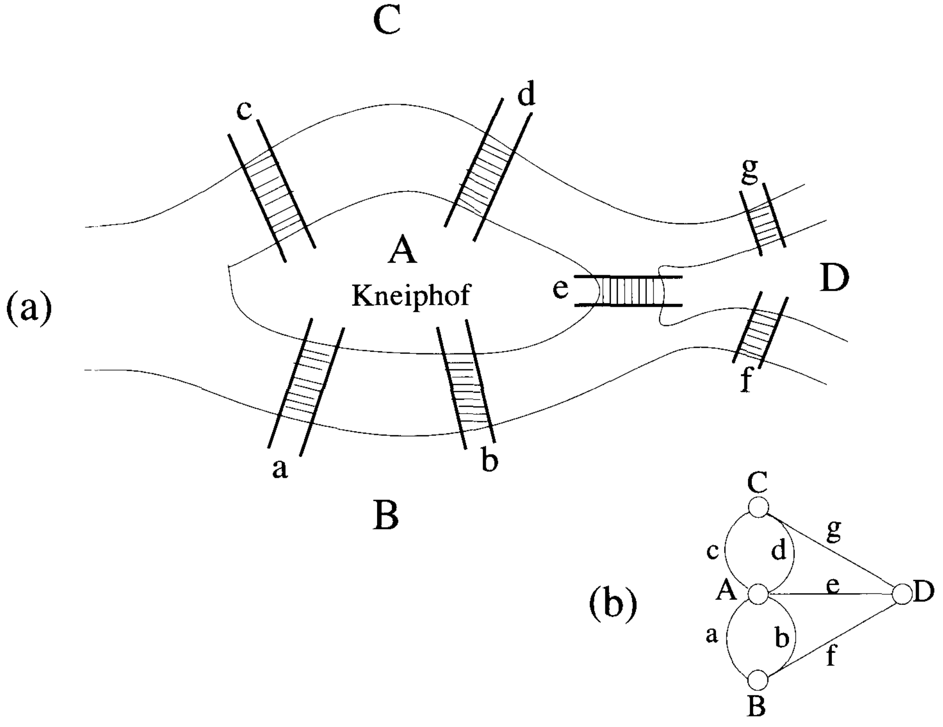


Figure 2.24 Section of the river Pregel in Königsberg and Euler's graph

head of the edge. Therefore, $\langle v, u\rangle$ and $\langle u, v\rangle$ represent two different edges. Figure 2.25 shows three graphs: G_1, G_2, and G_3. The graphs G_1 and G_2 are undirected; G_3 is directed.

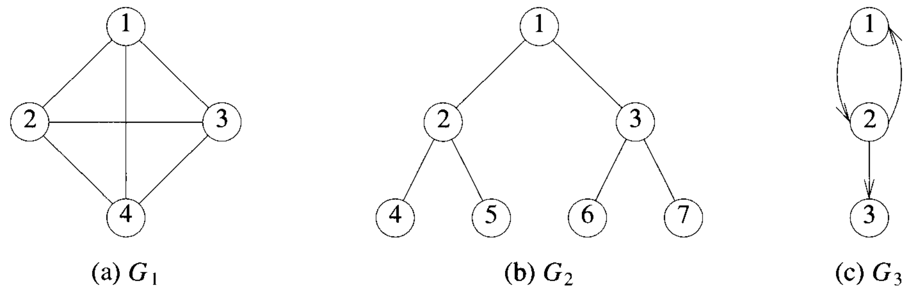


Figure 2.25 Three sample graphs

The set representations of these graphs are

$$\begin{array}{ll} V(G_1) = \{1,2,3,4\} & E(G_1) = \{(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)\} \\ V(G_2) = \{1,2,3,4,5,6,7\} & E(G_2) = \{(1,2),(1,3),(2,4),(2,5),(3,6),(3,7)\} \\ V(G_3) = \{1,2,3\} & E(G_3) = \{\langle 1,2\rangle,\langle 2,1\rangle,\langle 2,3\rangle\} \end{array}$$

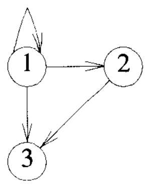
Notice that the edges of a directed graph are drawn with an arrow from the tail to the head. The graph G_2 is a tree; the graphs G_1 and G_3 are not.

Since we define the edges and vertices of a graph as sets, we impose the following restrictions on graphs:

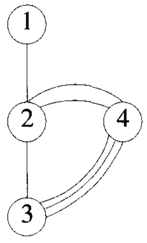
1. A graph may not have an edge from a vertex v back to itself. That is, edges of the form (v, v) and $\langle v, v\rangle$ are not legal. Such edges are known as *self-edges* or *self-loops*. If we permit self-edges, we obtain a data object referred to as a *graph with self-edges*. An example is shown in Figure 2.26(a).

2. A graph may not have multiple occurrences of the same edge. If we remove this restriction, we obtain a data object referred to as a *multigraph* (see Figure 2.26(b)).

The number of distinct unordered pairs (u, v) with $u \neq v$ in a graph with n vertices is $\frac{n(n-1)}{2}$. This is the maximum number of edges in any n-vertex, undirected graph. An n-vertex, undirected graph with exactly $\frac{n(n-1)}{2}$ edges is said to be *complete*. The graph G_1 of Figure 2.25(a) is the complete graph



(a) Graph with a self edge



(b) Multigraph

Figure 2.26 Examples of graphlike structures

on four vertices, whereas G_2 and G_3 are not complete graphs. In the case of a directed graph on n vertices, the maximum number of edges is $n(n-1)$.

If (u, v) is an edge in $E(G)$, then we say vertices u and v are *adjacent* and edge (u, v) is *incident* on vertices u and v. The vertices adjacent to vertex 2 in G_2 are 4, 5, and 1. The edges incident on vertex 3 in G_2 are (1,3), (3,6), and (3,7). If $\langle u, v\rangle$ is a directed edge, then vertex u is *adjacent to* v, and v is *adjacent from* u. The edge $\langle u, v\rangle$ is incident to u and v. In G_3, the edges incident to vertex 2 are $\langle 1,2\rangle$, $\langle 2,1\rangle$, and $\langle 2,3\rangle$.

A *subgraph* of G is a graph G' such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$. Figure 2.27 shows some of the subgraphs of G_1 and G_3.

A *path* from vertex u to vertex v in graph G is a sequence of vertices $u, i_1, i_2, \ldots, i_k, v$, such that $(u, i_1), (i_1, i_2), \ldots, (i_k, v)$ are edges in $E(G)$. If G' is directed, then the path consists of the edges $\langle u, i_1\rangle, \langle i_1, i_2\rangle, \ldots, \langle i_k, v\rangle$ in $E(G')$. The *length* of a path is the number of edges on it. A *simple path* is a path in which all vertices except possibly the first and last are distinct. A path such as (1,2), (2,4), (4,3), is also written as 1, 2, 4, 3. Paths 1, 2, 4, 3 and 1, 2, 4, 2 of G_1 are both of length 3. The first is a simple path; the second is not. The path 1, 2, 3 is a simple directed path in G_3, but 1, 2, 3, 2 is not a path in G_3, as the edge $\langle 3, 2\rangle$ is not in $E(G_3)$.

A *cycle* is a simple path in which the first and last vertices are the same. The path 1, 2, 3, 1 is a cycle in G_1 and 1, 2, 1 is a cycle in G_3. For directed graphs we normally add the prefix "directed" to the terms cycle and path.

In an undirected graph G, two vertices u and v are said to be *connected* iff there is a path in G from u to v (since G is undirected, this means there must also be a path from v to u). An undirected graph is said to be connected iff for every pair of distinct vertices u and v in $V(G)$, there is a path from u to v in G. Graphs G_1 and G_2 are connected, whereas G_4 of Figure 2.28 is not.

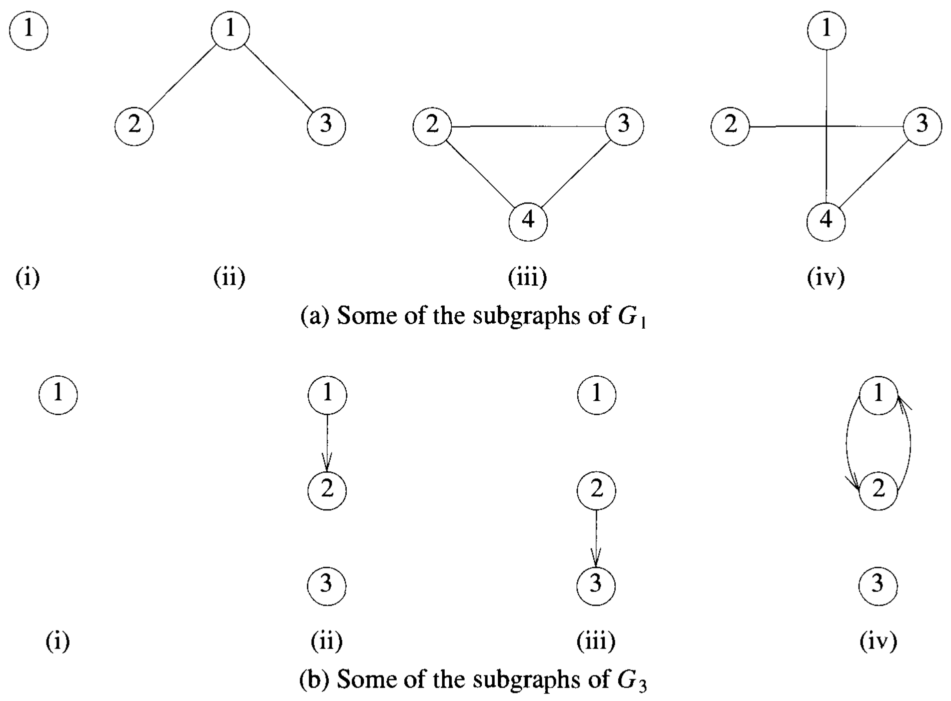

(i)
(ii)
(iii)
(iv)
(a) Some of the subgraphs of G_1
(i)
(ii)
(iii)
(iv)
(b) Some of the subgraphs of G_3

Figure 2.27 Some subgraphs

A *connected component* (or simply a component) H of an undirected graph is a *maximal* connected subgraph. By "maximal," we mean that G contains no other subgraph that is both connected and properly contains H. G_4 has two components, H_1 and H_2 (see Figure 2.28).

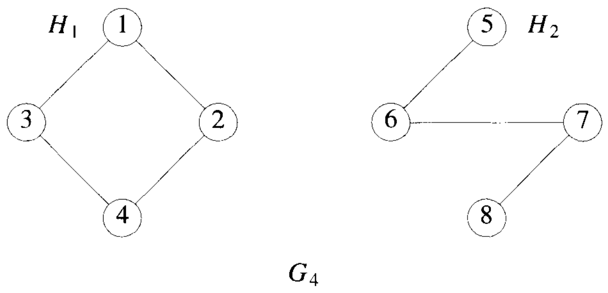


Figure 2.28 A graph with two connected components

A *tree* is a connected acyclic (i.e., has no cycles) graph. A directed graph G is said to be *strongly connected* iff for every pair of distinct vertices u and v in $V(G)$, there is a directed path from u to v and also from v to u. The graph G_3 (repeated in Figure 2.29(a)) is not strongly connected, as there is no path from vertex 3 to 2. A *strongly connected component* is a maximal subgraph that is strongly connected. The graph G_3 has two strongly connected components (see Figure 2.29(b)).

The degree of a vertex is the number of edges incident to that vertex. The degree of vertex 1 in G_1 is 3. If G is a directed graph, we define the *in-degree* of a vertex v to be the number of edges for which v is the head. The *out-degree* is defined to be the number of edges for which v is the tail. Vertex 2 of G_3 has in-degree 1, out-degree 2, and degree 3. If d_i is the degree of vertex i in a graph G with n vertices and e edges, then the number of edges is

$$e = \left(\sum_{i=1}^{n} d_i \right) /2$$

In the remainder of this chapter, we refer to a directed graph as a *digraph*. When we use the term graph, we assume that it is an undirected graph.

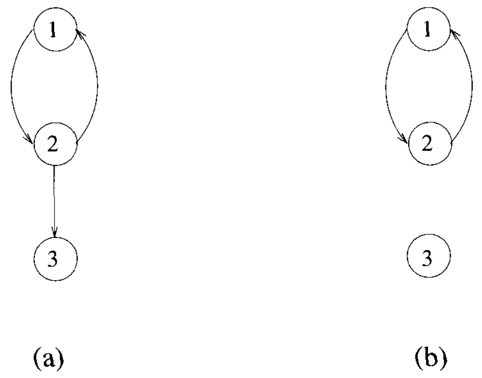


Figure 2.29 A graph and its strongly connected components

2.6.3 Graph Representations

Although several representations for graphs are possible, we study only the three most commonly used: adjacency matrices, adjacency lists, and adjacency multilists. Once again, the choice of a particular representation depends on the application we have in mind and the functions we expect to perform on the graph.

Adjacency Matrix

Let $G = (V, E)$ be a graph with n vertices, $n \geq 1$. The adjacency matrix of G is a two-dimensional $n \times n$ array, say a, with the property that $a[i, j] = 1$ iff the edge (i, j) ($\langle i, j \rangle$ for a directed graph) is in $E(G)$. The element $a[i, j] = 0$ if there is no such edge in G. The adjacency matrices for the graphs G_1, G_3, and G_4 are shown in Figure 2.30. The adjacency matrix for an undirected graph is symmetric, as the edge (i, j) is in $E(G)$ iff the edge (j, i) is also in $E(G)$. The adjacency matrix for a directed graph may not be symmetric (as is the case for G_3). The space needed to represent a graph using its adjacency matrix is n^2 bits. About half this space can be saved in the case of an undirected graph by storing only the upper or lower triangle of the matrix.

From the adjacency matrix, we can readily determine whether there is an edge connecting any two vertices i and j. For an undirected graph the degree of any vertex i is its row sum:

$$\begin{array}{c|cccc|} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 1 & 1 \\ 2 & 1 & 0 & 1 & 1 \\ 3 & 1 & 1 & 0 & 1 \\ 4 & 1 & 1 & 1 & 0 \\ \hline \end{array}$$

(a) G_1

$$\begin{array}{c|ccc|} & 1 & 2 & 3 \\ \hline 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 1 \\ 3 & 0 & 0 & 0 \\ \hline \end{array}$$

(b) G_3

$$\begin{array}{c|cccccccc|} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \hline 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 4 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 7 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 8 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ \hline \end{array}$$

(c) G_4

Figure 2.30 Adjacency matrices

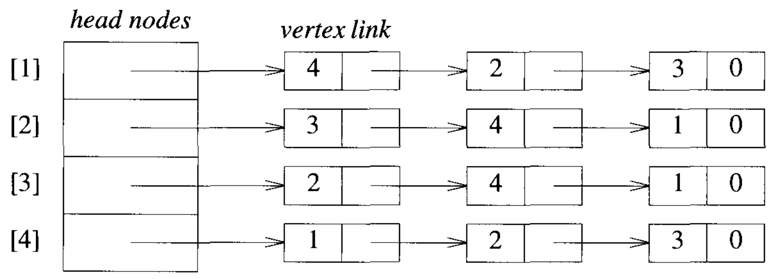
$$\sum_{j=1}^{n} a[i,j]$$

For a directed graph the row sum is the out-degree, and the column sum is the in-degree.

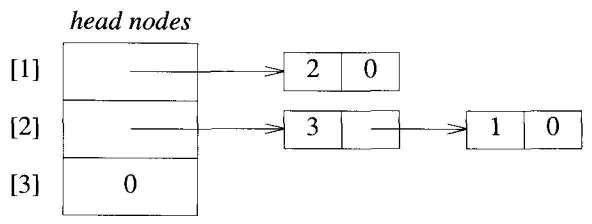
Suppose we want to answer a nontrivial question about graphs, such as How many edges are there in G? or Is G connected? Adjacency matrices require at least n^2 time, as $n^2 - n$ entries of the matrix (diagonal entries are zero) have to be examined. When graphs are sparse (i.e., most of the terms in the adjacency matrix are zero), we would expect that the former question could be answered in significantly less time, say $O(e+n)$, where e is the number of edges in G, and $e < \frac{n^2}{2}$. Such a speedup is made possible through the use of a representation in which only the edges that are in G are explicitly stored. This leads to the next representation for graphs, adjacency lists.

Adjacency Lists

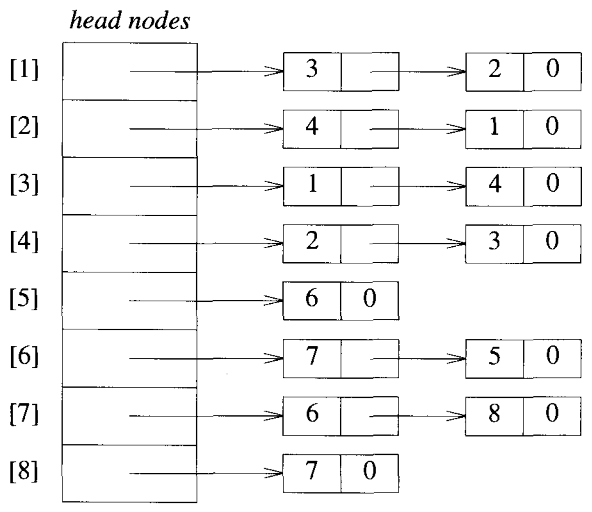
In this representation of graphs, the n rows of the adjacency matrix are represented as n linked lists. There is one list for each vertex in G. The nodes in list i represent the vertices that are adjacent from vertex i. Each node has at least two fields: *vertex* and *link*. The *vertex* field contains the indices of the vertices adjacent to vertex i. The adjacency lists for G_1, G_3,



(a) G_1



(b) G_3



(c) G_4

Figure 2.31 Adjacency lists

and G_4 are shown in Figure 2.31. Notice that the vertices in each list are not required to be ordered. Each list has a head node. The head nodes are sequential, and so provide easy random access to the adjacency list for any particular vertex.

For an undirected graph with n vertices and e edges, this representation requires n head nodes and $2e$ list nodes. Each list node has two fields. In terms of the number of bits of storage needed, this count should be multiplied by $\log n$ for the head nodes and $\log n + \log e$ for the list nodes, as it takes $O(\log m)$ bits to represent a number of value m. Often, you can sequentially pack the nodes on the adjacency lists, and thereby eliminate the use of pointers. In this case, an array *node* $[1 : n + 2e + 1]$ can be used. The *node*$[i]$ gives the starting point of the list for vertex i, $1 \leq i \leq n$, and *node*$[n + 1]$ is set to $n + 2e + 2$. The vertices adjacent from vertex i are stored in *node*$[i], \ldots,$*node*$[i + 1] - 1$, $1 \leq i \leq n$. Figure 2.32 shows the sequential representation for the graph G_4 of Figure 2.28.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
10	12	14	16	18	19	21	23	24	3	2	4	1	1	4	2	3	6	7	5	6	8	7

Figure 2.32 Sequential representation of graph G_4:
Array *node*$[1 : n + 2e + 1]$

The degree of any vertex in an undirected graph can be determined by just counting the number of nodes in its adjacency list. So, the number of edges in G can be determined in $O(n + e)$ time.

For a digraph, the number of list nodes is only e. The out-degree of any vertex can be determined by counting the number of nodes on its adjacency list. Hence, the total number of edges in G can be determined in $O(n + e)$ time. Determining the in-degree of a vertex is a little more complex. If there is a need to access repeatedly all vertices adjacent to another vertex, then it may be worth the effort to keep another set of lists in addition to the adjacency lists. This set of lists, called *inverse adjacency lists*, contains one list for each vertex. Each list contains a node for each vertex adjacent to the vertex it represents (see Figure 2.33).

One can also adopt a simpler version of the list structure in which each node has four fields and represents one edge. The node structure is

tail	head	column link for head	row link for tail

Figure 2.34 shows the resulting structure for the graph G_3 of Figure 2.25(c). The head nodes are stored sequentially.

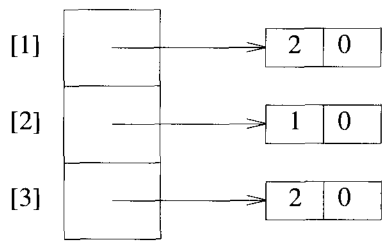


Figure 2.33 Inverse adjacency lists for G_3 of Figure 2.25(c)

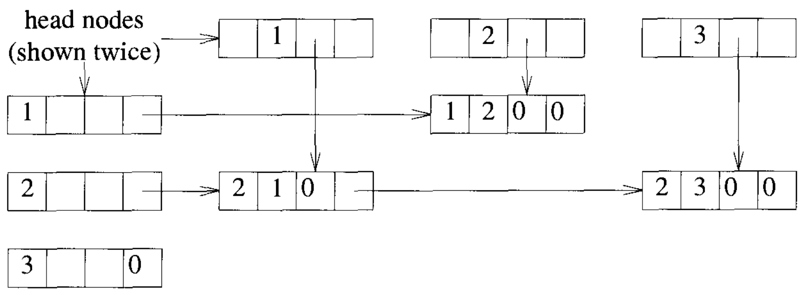


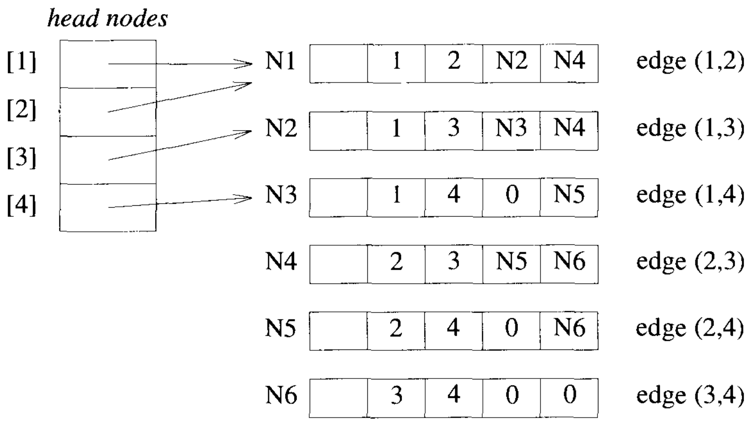
Figure 2.34 Orthogonal list representation for G_3 of Figure 2.25(c)

Adjacency Multilists

In the adjacency-list representation of an undirected graph, each edge (u, v) is represented by two entries, one on the list for u and the other on the list for v. In some applications it is necessary to be able to determine the second entry for a particular edge and mark that edge as having been examined. This can be accomplished easily if the adjacency lists are maintained as multilists (i.e., lists in which nodes can be shared among several lists). For each edge there is exactly one node, but this node is in two lists (i.e., the adjacency lists for each of the two nodes to which it is incident). The new node structure is

m	*vertex*1	*vertex*2	*list*1	*list*2

where m is a one-bit mark field that can be used to indicate whether the edge has been examined. The storage requirements are the same as for normal adjacency lists, except for the addition of the mark bit m. Figure 2.35 shows the adjacency multilists for G_1 of Figure 2.25(a).


head nodes
[1]
[2]
[3]
[4]
N1 | 1 | 2 | N2 | N4 | edge (1,2)
N2 | 1 | 3 | N3 | N4 | edge (1,3)
N3 | 1 | 4 | 0 | N5 | edge (1,4)
N4 | 2 | 3 | N5 | N6 | edge (2,3)
N5 | 2 | 4 | 0 | N6 | edge (2,4)
N6 | 3 | 4 | 0 | 0 | edge (3,4)

The lists are

vertex 1:	N1 → N2 → N3
vertex 2:	N1 → N4 → N5
vertex 3:	N2 → N4 → N6
vertex 4:	N3 → N5 → N6

Figure 2.35 Adjacency multilists for G_1 of Figure 2.25(a)

Weighted Edges

In many applications, the edges of a graph have weights assigned to them. These weights may represent the distance from one vertex to another or the cost of going from one vertex to an adjacent vertex. In these applications, the adjacency matrix entries $a[i, j]$ keep this information too. When adjacency lists are used, the weight information can be kept in the list nodes by including an additional field, *weight*. A graph with weighted edges is called a *network*.

EXERCISES

1. Does the multigraph of Figure 2.36 have an Eulerian walk? If so, find one.

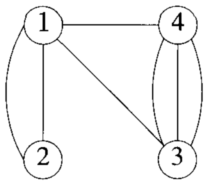


Figure 2.36 A multigraph

2. For the digraph of Figure 2.37 obtain
   (a) the in-degree and out-degree of each vertex
   (b) its adjacency-matrix representation
   (c) its adjacency-list representation
   (d) its adjacency-multilist representation
   (e) its strongly connected components

3. Devise a suitable representation for graphs so that they can be stored on disk. Write an algorithm that reads in such a graph and creates its adjacency matrix. Write another algorithm that creates the adjacency lists from the disk input.

4. Draw the complete undirected graphs on one, two, three, four, and five vertices. Prove that the number of edges in an n-vertex complete graph is $\frac{n(n-1)}{2}$.

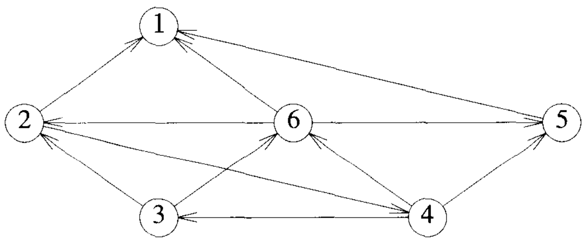


Figure 2.37 A digraph

5. Is the directed graph of Figure 2.38 strongly connected? List all the simple paths.

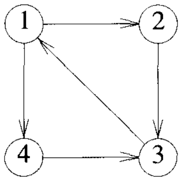


Figure 2.38 A directed graph

6. Obtain the adjacency-matrix, adjacency-list, and adjacency-multilist representations of the graph of Figure 2.38.

7. Show that the sum of the degrees of the vertices of an undirected graph is twice the number of edges.

8. Prove or disprove:

   If $G(V, E)$ is a finite directed graph such that the out-degree of each vertex is at least one, then there is a directed cycle in G.

9. (a) Let G be a connected, undirected graph on n vertices. Show that G must have at least $n - 1$ edges and that all connected, undirected graphs with $n - 1$ edges are trees.

   (b) What is the minimum number of edges in a strongly connected digraph with n vertices? What form do such digraphs have?

10. For an undirected graph G with n vertices, prove that the following are equivalent:
    (a) G is a tree.
    (b) G is connected, but if any edge is removed, the resulting graph is not connected.
    (c) For any two distinct vertices $u \in V(G)$ and $v \in V(G)$, there is exactly one simple path from u to v.
    (d) G contains no cycles and has $n - 1$ edges.

11. Write an algorithm to input the number of vertices in an undirected graph and its edges one by one and to set up the linked adjacency-list representation of the graph. You may assume that no edge is input twice. What is the run time of your procedure as a function of the number of vertices and the number of edges?

12. Do the preceding exercise but now set up the multilist representation.

13. Let G be an undirected, connected graph with at least one vertex of odd degree. Show that G contains no Eulerian walk.

2.7 REFERENCES AND READINGS

A wide-ranging examination of data structures and their efficient implementation can be found in the following:

Fundamentals of Data Structures in C++, by E. Horowitz, S. Sahni, and D. Mehta, Computer Science Press, 1995.

Data Structures and Algorithms 1: Sorting and Searching, by K. Mehlhorn, Springer-Verlag, 1984.

Introduction to Algorithms: A Creative Approach, by U. Manber, Addison-Wesley, 1989.

Handbook of Algorithms and Data Structures, second edition, by G. H. Gonnet and R. Baeza-Yates, Addison-Wesley, 1991.

Proof of Lemma 2.4 can be found in "Worst-case analysis of set union algorithms," by R. Tarjan and J. Van Leeuwen, *Journal of the ACM* 31; no. 2 (1984): 245–281.